A Global Name Service for a Highly Mobile Internetwork

Abhigyan Sharma Xiaozheng Tie Hardeep Uppal Arun Venkataramani
David Westbrook Aditya Yadav *
{abhigyan, xztie, hardeep, arun, westy, ayadav}@cs.umass.edu

School of Computer Science, University of Massachusetts Amherst

ABSTRACT

Mobile devices dominate the Internet today, however the Internet rooted in its tethered origins continues to provide 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 service 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 service that addresses this challenge. A key insight underlying Auspice is a *demand-aware* replica placement engine that intelligently replicates name records to provide low lookup latency, low update cost, and high availability. We have implemented a prototype of Auspice and compared it against several commercial managed DNS providers as well as state-of-the-art research alternatives, and shown that Auspice significantly outperforms both. We demonstrate proof-of-concept that Auspice can serve as a complete end-to-end mobility solution as well as enable novel context-based communication primitives that generalize name- or address-based communication in today's Internet.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: COMPUTER-COMMUNICATION NETWORKS—*Distribtued Systems*

Keywords

Mobility; distributed systems; network architecture

1. INTRODUCTION

"Mobile" has long arrived, but the Internet remains unmoved. Today, there is roughly one cellphone per human; the number of smartphones sold last year alone roughly equals the number of wired hosts on the Internet [28]; and the total traffic originated by mobiles is poised to approach that by wired devices [18]. However, the current Internet continues to operate as it did when dominated by tethered hosts, simply ignoring frequent endpoint mobility.

Today, an application developer 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 smartphone notification systems, playback video, or cloud storage 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 (as popular VoIP apps do today). The lack of intrinsic support for mobility means that developers are forced to redundantly develop and maintain common-case functionality. Furthermore, we are paying an unknowable price in terms of long-term growth and innovation by strait-jacketing communication initiation to be unidirectional.

Many before us have criticized the Internet architecture's poor support for mobility as well as multihoming [33, 8, 24, 54]. 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 could be employed to seamlessly handle mid-connection mobility.

But a key missing element from this package today is a distributed name resolution infrastructure that can scale to orders of magnitude higher update rates than envisioned when DNS was created. To appreciate the envisioned scale, consider tens of billions of mobile identifiers changing network addresses at least tens of times per day. DNS's heavy reliance on TTL-based caching, a key strength recognized by its creators, researchers, and operators alike, poses a significant handicap by increasing update propagation delays, load on name servers, and overall client-perceived latency. 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 [7, 47]. A less widely noted limitation of DNS is its reliance on hierarchical names for scaling via federation and its single root of trust, which constrains mobile applications from selecting arbitrary application-specific names (as elaborated in §2.2 and §3.2).

*Authors ordered alphabetically by last name.

SIGCOMM'14, August 17–22, 2014, Chicago, IL, USA.

ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626331.

Our position is that seamless support for mobility requires a logically centralized 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 helps address this challenge. Compared to today's ICANN/DNS-based approach, our approach cleanly separates name resolution from adjudication and certification issues (§3.2). Auspice is also deployable as a managed DNS provider in today's Internet; compared to them, a key strength of Auspice is a *demand-aware* replica placement engine that significantly reduces the *time-to-connect* to mobile destinations in a cost-effective manner. Under light load, Auspice's demand-aware replica placement aggressively uses available resources to massively replicate name records, while under heavy load, it carefully controls the number and choice of replica locations based on the read-write patterns and pockets of high demand for each name.

We have implemented a prototype of Auspice as a geo-distributed key-value store to serve as a flexible name resolution service for the current Internet as well as several "future" Internet or endpoint architectures such as MobilityFirst[54], HIP[33], or XIA[31]. We have extensively evaluated Auspice using a combination of Planetlab, emulation clusters, and Amazon EC2. Our contributions are as follows.

1. A case for a global name service as an indispensable part of any Internetwork design with intrinsic support for high mobility (§2).
2. Auspice, a scalable, geo-distributed, federated global name service that significantly reduces the time-to-connect under any given resource constraints despite high mobility and arbitrary endpoint identifiers (§3,§4).
3. A proof-of-concept demonstration of intrinsic support for—*(i)* all four types of endpoint mobility; *(ii)* novel context-aware delivery primitives that generalize name- or address-based communication—over the current Internet as well as MobilityFirst [54] (§4.3).
4. Comparison against several best-of-breed managed DNS services showing that Auspice's demand-aware approach significantly lowers time-to-connect and/or update cost even for today's (hardly mobile) domain names (§4.4).

To provide a historical perspective, 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 arose in response to the rapidly increasing file size and distribution costs. Mockapetris and Dunlap [43] point to TTL-based caching to reduce load and response times as a key strength, noting that "*the XEROX system* [Grapevine [51]] *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 active replication (§2.2) in Auspice in order to address the challenges of mobility, a concern that wasn't particularly pressing in the 80s. Compared to classical systems like Grapevine or ClearingHouse, Auspice enables support for automated *demand-aware* replica placement for *arbitrary names* (using several modern design elements such as consensus, the key-value abstraction, self-certifying names, consistent hashing, etc). Auspice, through its support for context-aware delivery, is also a step towards addressing some of the challenges to which Lampson alludes on representing "descriptive names" [40].

2. CASE FOR A GLOBAL NAME SERVICE

Given the huge body of prior work specifically on mobility as well as more broadly on Internet architecture, it is natural to begin by asking: Is a global name resolution service critical to handling mobility if we had the luxury of refactoring Internet naming and routing from a clean slate?

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.

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 [48] or a flat identifier's consistent-hash location in i3 [53]) 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 [33], AIP[12], XIA[31], or MobilityFirst[4]) or an identifier in LISP [8] or HAIR [24] to either an IP address [33], a self-certifying network identifier [12, 31, 4], or a hierarchical locator [24] that encodes routing information. Global name-to-address resolution schemes also subsume DHT-based Internet architectures such as LNA [14, 56] as well as resolution systems like CoDoNS [49] 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 [52, 13, 26] that end-to-end connection migration, i.e., bilaterally without relying on an external service, suffices to migrate connections efficiently when endpoints move one at a time, but 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 common in 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.

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

Name-based routing schemes in the ideal have a tantalizing intellectual lure—to seamlessly handle mobility by routing directly over names with no resolution step—but are marred by several fundamental and practical challenges. First, name-based routing approaches can support seamless mobility only if routing update propagation delays are on the order of milliseconds, a daunting challenge given that inter-domain routing can take several minutes to converge today. Second, theoretical results on compact routing [36] 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 [17]) 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 [29, 35, 32] (e.g., NDN-style [32] names such as `/umass/phone42/call3/frame7`), frequent mobility still poses a challenge as routers would have to maintain special forwarding entries for "displaced names", i.e., names that move from their hierarchically organized namespace (say, from `/umass` to `/comcast` in this example) for longest-prefix matching to work correctly. That is, 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, a conjecture that has recently been empirically reinforced by Gao et al. [27].

Summary. Our position is that a global name-to-address resolution service is critical for 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 [4, 31]). Perhaps the most compelling argument for global name-to-address resolution is our decades of familiarity with DNS and the Internet; handling mobility would be a drop-in replacement to DNS provided we address the challenge of building a distributed system that scales to billions of devices making many updates a day and yet returns up-to-date responses within milliseconds.

2.2 Limitations of DNS

What specific design traits of DNS make it poorly suited for mobile applications? The first two traits below limit its scalability with respect to the rate of endpoint mobility, and the third limits its scalability with respect to the size of the namespace if applications were to have the luxury of using arbitrary (but fixed) names.

(1) *TTL-based caching:* TTL-based caching is the single-most important mechanism for DNS's scalability; caching not only helps DNS sustain essentially arbitrarily high *lookup load* but also dramatically reduces client-perceived *lookup latency* for cache hits. However, caching is ineffective when TTLs are near-zero, as would have to be the case under high mobility, causing both increased load on name servers and higher client-perceived latencies. Caching is also less effective if lookups are distributed relatively uniformly, as could be the case with mobile device names, unlike lookups for today's domain names that are highly skewed [34, 45].

(2) *Static placement:* Authoritative DNS servers are essentially rendezvous points that allow a mobile endpoint to inform potential correspondents of its current location(s). In order to reduce the time-to-connect, authoritative servers must be located close to potential correspondents. However, authoritative server locations today are static, either close to a mobile endpoint's "home" location or a pre-packaged set of geo-distributed locations provided by a managed DNS provider. Engineering a scalable geo-distributed system that can dynamically move object replicas in a demand-aware manner is nontrivial and real-world examples of such systems have only recently begun to emerge [19].

(3) *Hierarchical names:* The hierarchical structure of DNS names is key to leveraging *federation* to scale to an arbitrary number of names by delegating different portions of the name space (or zones) to different organizations. For example, root name servers today only have to maintain state for a small number of top-level domain names. In contrast, arbitrary or flat names, e.g., "`JohnSmith3142's watch`" can not be supported in DNS while retaining the scaling benefits of federation as the root name servers would have to maintain nonzero state, e.g., at least the authoritative name server(s) and the DNSSEC key of a name, for essentially all names. Our position is that the design of a general-purpose global name service must not restrict the structure of names as names carry application-specific semantics; in §4.3.3, we show examples of novel context-aware communication primitives that are feasible with unrestricted names.

Our approach to address the first two issues above relies on *active* and *demand-aware* replication: (1) Active replication significantly reduces (but does not eliminate) the reliance on passive caching; (2) Demand-aware replication ensures that active replicas of a name record are accessible close to clients querying the name, so as to reduce the overall time-to-connect. A glib but pedagogically helpful way to highlight the difference from DNS is that, in the extreme case, our approach can create an active replica of a name record near every DNS local name server that stores a passively cached copy today. Our approach addresses the third issue above by cleanly separating resolution of names from adjudication and certification. We explain our approach in detail next.

3. Auspice DESIGN & IMPLEMENTATION

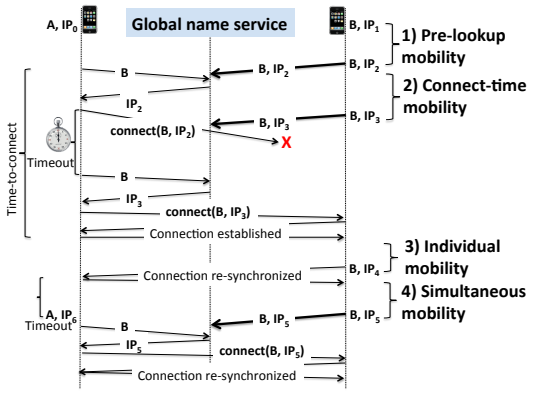


Figure 1: Four kinds of mobility—(1) *pre-lookup*, (2) *connect-time*, (3) *individual*, (4) *simultaneous*—three of which require a global name service.

Our envisioned GNS enables endpoint mobility as shown in Figure 1. An endpoint A initiates communication with another endpoint B by querying the GNS for B's current addresses and connecting to one of them, thereby enabling *pre-lookup* mobility. If B moves after A's query but before before a connection has been mutually established via a three-way handshake (*connect-time* mobility), A times out and reverts back to the GNS. After a connection has been established, if either endpoint moves one at a time (*individual* mobility), it can re-synchronize the connection with a bilateral three-way handshake without relying upon the GNS (noting however that router-level late-binding proposals relying on a GNS-like infrastructure have also been proposed [44, 4]). If an endpoint moves mid-connection after the other endpoint has moved but before it could re-synchronize the connection (*simultaneous* mobility), one or both endpoint(s) eventually query the GNS and re-synchronize the connection.

3.1 Design goals

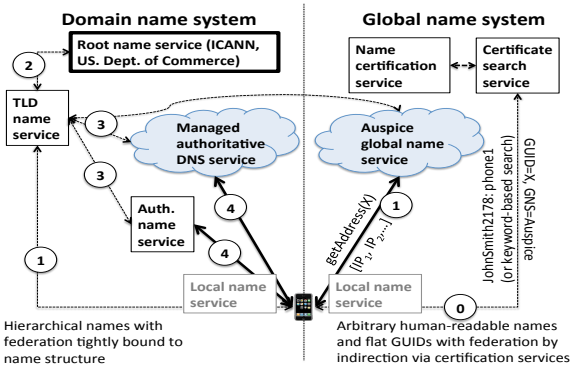


Figure 2: DNS vs. GNS: Auspice can be deployed as a managed DNS provider today (left) or as a GNS provider that provides resolution service for its customer GUIDs (right). Name certification services bind a human-readable name to a GUID and its GNS provider, and certificate search services can help index and distribute certificates from all certification services. Solid (dotted) lines represent frequent (infrequent) query paths for a given mobile destination. Except for the tightly controlled DNS root service, all services above are designed to be purveyed competitively.

Much of the envisioned functionality of a GNS as above boils down to one over-arching distributed systems challenge: *any principal–endpoint or router–should get the look and feel of a high-availability name service that is nearby (≈ few milliseconds) and rapidly returns up-to-date responses.* A more precise breakdown of goals is as follows.

(1) Time-to-connect performance: The design must ensure low latencies for name lookups to return up-to-date values, which determines the *time to connect* to a destination when the value being queried for is an address like above.

(2) Resource cost: The design must ensure low replication cost. A naive way to minimize lookup latencies is to replicate every name record at every possible location, however high mobility means high update rates, so the cost of pushing each update to every replica would be prohibitive. Worse, load hotspots can actually degrade lookup latencies.

(3) High availability: The design must ensure resilience to node failures including outages of entire datacenters; by consequence, it should also prevent crippling load hotspots.

(4) Security: The design must be robust to malicious users attempting to hijack or corrupt name records. The design must support flexible access control policies to ensure the desired level of privacy of name records.

(5) Federation: The design must allow different name service providers to co-exist and for users to freely choose one or more preferred providers.

(6) Extensibility: The design must be agnostic to how names, addresses, and resolution policies are represented by a future Internetwork. In particular, it should support flat names and a rich set of attributes and resolution policies for multi-homed mobility (e.g.,"prefer WiFi to cellular"), etc.

3.2 Design overview

To address the above goals, the Auspice GNS is designed as a massively geo-distributed key-value store. The geo-distribution is essential to the latency and availability goals while the key-value API enables extensibility. Each *name record* in Auspice is associated with a globally unique identifier (GUID) that is the record's primary key. A name record contains an associative array of key-value pairs, wherein each key K_i is a string and the value V_i may be a string, a primitive type, or recursively a key-value pair, as shown below.

GUID $\mid K_1, V_1 \mid K_2, V_2 \mid \cdots$

The GUID is a self-certifying identifier computed as a compact one-way hash of a public key. Each name record is aliased to one or more globally unique human-readable names that are bound to the GUID by a certificate supplied by one or more *name certification service(s)* (NCS). Loosely speaking, the human-readable name is analogous to a DNS domain name and a name record to a zone file, but with the following important differences.

Security. As shown in Fig. 2, to initiate communication with a destination Y, an endpoint X must first obtain a certificate of the form $[\texttt{JohnSmith2178:Phone1}, Y, P]_{K^-}$ that binds the human-readable name to the GUID Y and its GNS provider P, and is signed by the private key K^- of an NCS that X trusts. A certificate search service (e.g., a search engine or ISP) can help index certificates from different NCSes, and help X find a certificate from a trusted NCS, and even find the human-readable name based on keywords.

Federation. Unlike ICANN and root DNS servers that respectively act as a single name adjudication authority and root of trust for certification, our approach decentralizes trust across different NCS providers, potentially allowing endpoints to use quorum-based approaches to resolve conflicting name certificates. More importantly, our federation approach allows endpoints to select arbitrary human-readable names and NCS providers unlike DNS that restricts domain names to be hierarchical and federation and the DNSSEC keychain to strictly follow the name structure. An inevitable implication of decentralizing trust is that two endpoints can communicate securely only if they share a trusted NCS provider, but this change we argue is preferable to and a strict generalization of the single-root-of-trust model that some perceive as arbitrary [10, 9].

Extensibility. Our design cleanly separates the GNS provider's resource-intensive responsibility of name resolution under high mobility from the slow-changing certification process. It also allows for the GNS provider to be deployed today as a managed authoritative DNS provider (Fig. 2) with the DNSSEC key deriving the GUID. Finally, the key-value store API enables an extensible name record represen-

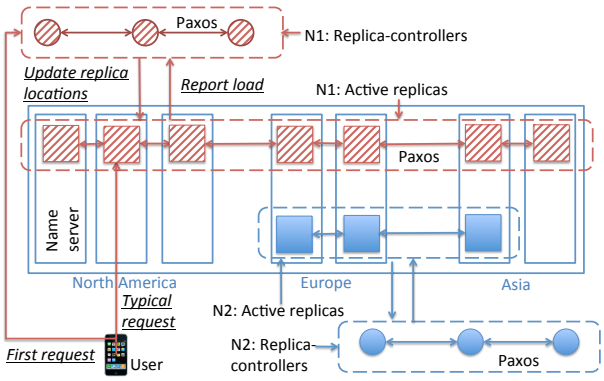


Figure 3: Geo-distributed name servers in Auspice. Replica-controllers (logically separate from active replicas) decide placement of active replicas and active replicas handle requests from end-users. N1 is a globally popular name and is replicated globally; name N2 is popular in select regions and is replicated in those regions.

tation. By default, each top-level key has associated read and write ACLs that could either be a blacklist or whitelist of GUIDs that respectively have read or write access. For example, a name record for GUID `X` that helps context-aware delivery or multihoming policies (detailed in §4.3.3) is below.

```
{X: {IPs:[{IP: 23.55.66.43, plan: Unlimited}, {IP:
62.44.65.75, plan: Limited}], geoloc: {[lat,long],
readWhitelist:[Y,Z]}, multihome_policy: Unlimited}}
```

3.3 Auspice's geo-distributed design

Next, we explain how Auspice achieves the first three design goals. 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 for the name, (2) the geo-distribution of requests for the name, and (3) the aggregate request load across all names.

Figure 3 illustrates the placement engine. Each name is associated with a fixed number, F, of *replica-controllers* and a variable number of *active replicas* of the corresponding name record. The name's replica-controllers are computed using consistent hashing to select F consecutive or otherwise deterministic nodes along the ring *onto* which the hash function maps names and nodes. The replica-controllers are responsible only for determining the number and locations of the active replicas, and the actives replicas are responsible for maintaining the actual name record and processing client requests. The replica-controllers implement a replicated state machine using Paxos [38] in order to maintain a consistent view of the current set of active replicas.

A name's replica-controllers compute its active replica locations in a *demand-aware* manner. This computation proceeds in epochs as follows. At creation 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 by that replica. Here, *regions* partition users into non-overlapping groups that capture locality, e.g., IP prefixes or a geographic partitioning based on cities; and the *load report* is a spatial vector of request rates as seen by the replica. The replica-controllers aggregate these load reports to obtain a concise spatial distribution of all requests for the name.

3.3.1 Demand-aware replica placement

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, practical algorithms, we defer the details of the optimization approach [1], using it only as a benchmark in small-scale experiments with 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 $(F + \beta r_i/w_i)$, where r_i and w_i are the lookup and update rates of name i; F is the minimum number of replicas needed to meet the availability goal (§3.1); and β is a replication control parameter that is automatically determined by the system so as to trade off latency benefits of replication 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 preset threshold utilization fraction μ. 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

$$\mu C = \sum_i r_i + \sum_i (F + \beta \frac{r_i}{w_i}) w_i \qquad (1)$$

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 as above, replica-controllers compute the locations of active replicas for name i as follows. Out of the $F + \beta r_i/w_i$ total replicas, a fraction ν of replicas are chosen based on locality, i.e., replica-controllers use the spatial vector of load reports to select $\nu(F + \beta r_i/w_i)$ name servers that are respectively the closest to the top $\nu(F + \beta r_i/w_i)$ regions sorted by demand for name i. The remaining $(1 - \nu)(F + \beta r_i/w_i)$ are chosen randomly without repetition. The locality-based 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 chose them based on round-trip latency alone, but we found that adding load-induced latencies in this step (in addition to choosing the remaining replicas randomly) ensures better load balance and lowers overall client-perceived latency. Our current prototype and system experiments fix the random perturbation knob ν to 0.5. We have since developed a slightly modified placement scheme that relieves the

administrator from setting ν manually, automatically balancing locality-awareness and load to ensure low latencies [1]. Thus, an administrator need only specify F and μ based on fault tolerance and aggressiveness of capacity utilization.

3.3.2 Client request routing

A client request is routed from an end-host to a suitable name server as follows. The set of all name servers in an Auspice instance is known to each member name server and can be obtained from a well-known location. 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 consistent hashing to determine the replica-controllers for that name and sends the request to the closest replica-controller. The replica-controller 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 until they change in some future epoch.

Network latency as well as server-load-induced latency help determine the closest replica at a local name server. Each local name server maintains an estimate of the round-trip latency to all name servers using infrequent pings; an (as yet unimplemented) optimization to reduce the overhead of all-to-all pings is to use coordinate embedding, geo-IP, or measurement-driven techniques [42]. To incorporate 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. If a lookup request sent to a name server times out, the local name server infers that either the server or network route is congested, and it multiplicatively increases its latency estimate to that name server by a fixed factor. Thus, if multiple lookups sent to a name server time out, the estimated latency shoots up and the local name server stops sending requests to that name server, which effectively acts as a more agile load-balancing policy in the request routing plane (complementing the replica placement plane above that operates in coarser-grained epochs).

3.3.3 Consistency with static replication

As a global name-to-address resolution service, Auspice must at least ensure this eventual consistency property: *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*; "eventually" means that there are no updates to a name record and no replica failures for sufficiently long. Violating this property means that a mobile client may be persistently unreachable even though it is no longer moving (updating addresses).

With a static set of replicas, it is straightforward to support this property. A replica receiving a client update need only record the write in a persistent manner locally, return a commit to the client, and lazily propagate the update to other active replicas for that record. Lazy propagation is sufficient to ensure that all replicas eventually receive every update committed at any replica, and a deterministic reconciliation policy, e.g., as in Dynamo [21], suffices to ensure that concurrent updates are consistently applied across all replicas. Temporary divergence across replicas under failures can be shortened by increasing durability, i.e., by recording the update persistently at more replicas before returning a commit to the client. The additional "single-writer" clause is satisfied simply by incorporating a client-local timestamp in the deterministic reconciliation policy.

Total write ordering. As Auspice is designed to be an expressive name service with sophisticated attributes, it may be useful in some scenarios 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. Ensuring a total ordering of all updates to a name is a stronger property than eventual consistency, calling for a state-machine approach, which Auspice supports as an option.

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 for that name). Each update is forwarded to the active replica that is elected as the Paxos coordinator that, under graceful execution, first gets a majority of replicas to accept the update number and then broadcasts a commit. Total write ordering of course implies that updates can make progress only when a majority of active replicas can communicate with each other while maintaining safety (consistent with the so-called CAP dilemma).

3.3.4 Consistency with replica reconfiguration

With a dynamic set of replicas as in Auspice, achieving eventual consistency is straightforward, as it suffices if a replica recovering from a crash lazily propagates all pending writes to a name to its *current* set of active replicas as obtained from any of the consistently-hashed replica-controllers for the name. However, satisfying the (optional) total write order property above is nontrivial.

To this end, we have designed a two-tier reconfigurable Paxos system that involves explicit coordination between the consensus engines of the replica-controllers and active replicas. Reconfiguration is accomplished by a replica-controller issuing and committing a stop request that gets committed as the last update of the current active replica group. The replica-controller subsequently initiates the next group of active replicas that can obtain the current record value from any member of the previous group. This design shares similarities with Vertical Paxos [39], however we were unable to find existing implementations or even reference systems using similar schemes, so we had to develop it from scratch. The details of the reconfiguration protocol are here [1].

3.3.5 Scalability: A performance-cost analysis

Cost. Auspice's replica placement scheme (Eq. 1) is designed to use a fraction μ of system-wide resources so as to make at least F and at most M replicas of each name, where M is the total number of name server locations. Thus, at light load, Auspice may replicate every name at every location, while under heavy load, it may create exactly F replicas for all but the most popular names. In the common case, a lookup involves one request and response between a local name server and an active replica; an update involves $\approx$ thrice (twice) as many messages as the number of active replicas with total write ordering (eventual) consistency.

Performance. The worst-case time-to-connect latency for a name i depends on the lookup latency l_i, the update rate w_i, the worst-case update propagation latency d_i, i.e., the time for all active replicas to receive an update, and the connect timeout T (Fig. 1), as follows [1].

$$\text{TTC}_i \quad = \quad l_i[1 + (e^{w_i d_i} - 1)(1 + \frac{T}{l_i})] \tag{2}$$

Thus, the time-to-connect increases with (1) the lookup latency l_i that in turn improves with demand-aware replication; (2) the update rate w_i and update propagation delay d_i that in combination determine the likelihood of obtaining a stale response, noting that the latter increases with more aggressive replication; and (3) the connect timeout T that is at most the default transport-layer timeout (e.g., a few seconds for TCP) and potentially as low as the round-trip delay between the connecting client and the destination being connected to if the destination network is capable of generating an "no route to host" error message.

TTLs. The above analysis implicitly assumes near-zero TTLs. With a nonzero TTL_i for name i, the worst-case time-to-connect can be approximated as [1].

$$\begin{aligned} \text{TTC}_i \quad \simeq \quad & \tau_i \, \frac{(r_i + w_i + 1/\text{TTL}_i + r_i w_i \text{TTL}_i)}{(1 + r_i \text{TTL}_i)(r_i + w_i + 1/\text{TTL}_i)} \\ & + \frac{T \; r_i \; w_i}{(r_i + 1/\text{TTL}_i)(r_i + w_i + 1/\text{TTL}_i)} \end{aligned} \tag{3}$$

where τ_i above is TTC_i (with a 0 TTL) as in Eq 2. Thus, a long TTL is meaningful only if the update rate w_i is low; if so, a carefully chosen TTL can reduce the load on the system as well as the client-perceived time-to-connect; if not, a long TTL can inflate the time-to-connect by the connect timeout T (= the second term above for $w_i \gg r_i$ and high TTL_i).

Comparison to DNS. All of the above analyses are applicable also to geo-replicated managed DNS providers were they to employ Auspice's demand-aware replication approach. The main difference between Auspice and today's managed DNS providers that rely on simplistic static replica placement schemes is in the lookup latency l_i achieved for any given resource cost; we evaluate this performance-cost tradeoff extensively in our experiments (§4.2 and §4.4).

3.3.6 Implementation status

We have implemented Auspice as described in Java with 28K lines of code. We have been maintaining an alpha deployment for research use for many months across eight EC2 regions. We have implemented support 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 what Auspice provides.

We have developed a simple NCS as a proof of concept, which through a web portal (`http://gns.name`) or a command-line console allows a user to bind a self- or system-selected GUID to a human-readable name that is simply an email address, i.e., our proof-of-concept NCS is a trivial CA that relies on email-based identity verification. Clients currently have to use a custom Auspice developer library to perform lookups and updates or custom socket library, msocket (§4.3), for end-to-end mobility features. We have also developed a simple proxy to translate between BIND and Auspice's wire-line protocol so as to interoperate with DNS.

4. EVALUATION

Our evaluation seeks to answer the following questions: (1) How well does Auspice's design meet its performance, cost, and availability goals compared to state-of-the-art alternatives under high mobility? (2) Can Auspice serve as a complete, end-to-end solution for mobility and enable novel communication abstractions? (3) How does Auspice's cost-performance tradeoff compare to best-of-breed managed DNS services for today's (hardly mobile) domain name workloads?

4.1 Experimental setup

Testbeds: We use geo-distributed testbeds (Amazon EC2 or Planetlab) or local emulation clusters (EC2 or a departmental cluster) depending upon the experiment's goals.

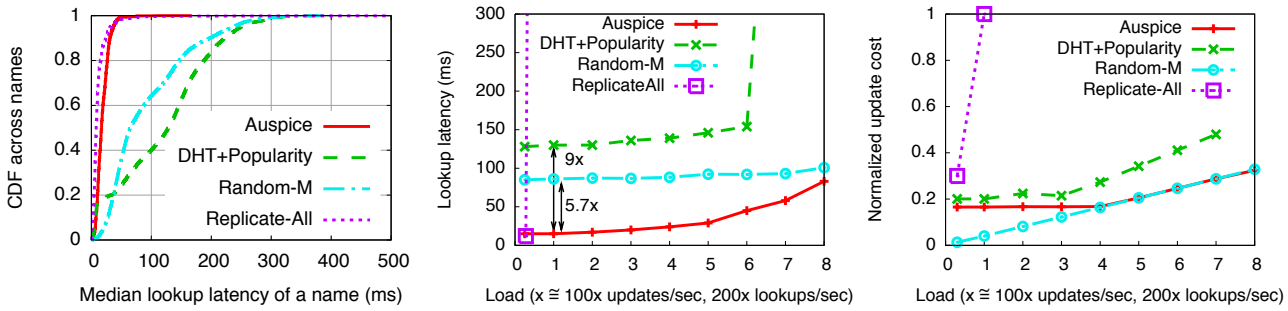
Workload: There is no real workload today of clients querying a name service in order to communicate with mobile devices frequently moving across different network addresses, both because such a name service does not exist and mobile devices do not have publicly visible IP addresses. So we conduct an evaluation using synthetic workloads for device names (§4.2), but to avoid second-guessing future workload patterns, we conduct a comprehensive sensitivity analysis against all of the relevant parameters such as the read rate, write rate, popularity, and geo-locality of demand [1].

The following are default experimental parameters for *device names*. 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 single device name is uniformly distributed between 0.5–1.5× the average lookup rate; the update rate is similarly distributed and drawn independently.

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 the 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 10% of the local name servers, the first of which is picked randomly and the rest are the ones geographically closest to it. We pick the geo-locality $g = 0.75$ for device names, i.e., the top 10% of regions in the world will account for 75% of requests, an assumption that is consistent with the finding that communication and content access exhibits a high country-level locality [37], and is consistent with the measured geo-locality (below) of service names today.

In addition to device names, *service names* constitute a small fraction (10%) of names and are intended to capture domain names like today with low mobility. Their lookup rate (or popularity) distribution and geo-distribution are used directly from the Alexa dataset [2]. Using this dataset, we calculated the geo-locality exhibited by the top 100K 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 devices).

Replication schemes compared: Auspice uses the replica placement strategy as described in §3 with the default parameter values $F = 3, \mu = 0.7, \nu = 0.5$. We compare Auspice against the following: (1) **Random-M** replicates each name at three random locations; (2) **Replicate-All** replicates all names at all locations; (3) **DHT+Popularity** replicates names using consistent hashing with replication similar to Codons[49]. The number of replicas is chosen based on the popularity ranking of a name and the location of replicas is decided by consistent hashing. The average hop count in Codons's underlying Beehive algorithm is set so that it creates the same average number of replicas as



(a) Lookup latencies (load = 0.3) (b) Lookup latency vs. load (c) Update cost vs. load

Figure 4: Auspice has up to 5.7× to 9× lower latencies than Random-M and DHT+Popularity reps. (4(b)). A load of 1 means 200 lookups/sec and 100 updates/sec per name server. Replicate-All peaks out at a load of 0.3 while Auspice can sustain a request load of up to 8 as it carefully chooses between 3 and 80 replicas per name.

Auspice for a fair comparison. All schemes direct a lookup to the closest available replica after the first request.

4.2 Evaluating Auspice's replica placement

We conduct experiments in this subsection on a 16-node (each with Xeon 5140, 4-cores, 8 GB RAM) departmental cluster, wherein each machine hosts 10 instances of either nameservers or local nameservers so as to emulate an 80-nameserver Auspice deployment. We instrument the instances so as to emulate wide-area latencies between any two instances that correspond to 160 randomly chosen Planetlab nodes. We choose emulation instead of a geo-distributed testbed in this experiment in order to obtain reproducible results while stress-testing the load-vs.-response time scaling behavior of various schemes given identical resources.

4.2.1 Lookup latency and update cost

How well does Auspice use available resources for replicating name records? To evaluate this, we compare the lookup latency of schemes across varying load levels. A machine running 10 name servers receives on average 2000 lookups/sec and 1000 updates/sec at a load = 1. For each scheme, load is increased until 2% of requests 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 DHT+Popularity pre-compute the placement at the start of the experiment based on prior knowledge of the workload.

Figure 4(a) shows the distribution of median lookup latency across names at the smallest load level (load = 0.3). Figure 4(b) shows load-vs-lookup latency curve for schemes, where "lookup latency" refers to the mean of the median lookup latencies of names. Figure 4(c) shows the corresponding mean of the distribution of update cost across names at varying loads; the update cost for a name is 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 4(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 the minimum of replicas Auspice must make for ensuring fault tolerance (resp. 80 and 3 here). *Random-M* can sustain a high request load (Fig. 4(b)) due to its low update costs, but its lookup latencies are higher as it only creates 3 replicas randomly.

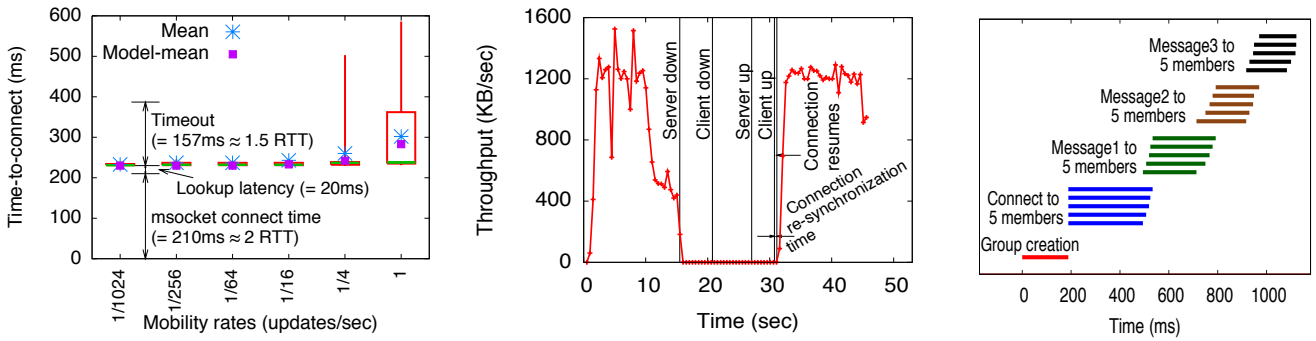
Auspice has 5.7×−9× lower latencies over Random-M and DHT+Popularity respectively (Figure 4(b), load=1). This is because it places a fraction of the replicas close to pockets of high demand unlike the other two. At low to moderate loads, servers have excess capacity than the minimum needed for fault tolerance, so Auspice creates as many replicas as it can without exceeding the threshold utilization level (Eq. 1), thereby achieving low 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 Random-M have equal update costs for loads ≥ 4 (Figure 4(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 Random-M.

DHT+Popularity has higher lookup latencies as it replicates based on lookup popularity alone and 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. DHT+Popularity 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.

DHT+Popularity incurs higher update costs than Auspice even though both schemes create nearly equal numbers of replicas at every load level. This is because DHT+Popularity 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, DHT+Popularity can not sustain as high a request load as Auspice.

4.2.2 Update latency, update propagation delay

The *client-perceived update latency*, i.e., the time from when when a client sends an update to when it receives a confirmation. These numbers are measured from the experiment in §4.2.1 for load=0.3. The median and 90th percentile update latency for Auspice with total write ordering is 284ms and is comparable to other schemes. A request, after arriving an active replica, takes four one-way network delays (two rounds) to be committed by Paxos. The median update latency is a few hundred milliseconds for all schemes as it is dominated by update propagation delays.



(a) E2E time-to-connect (b) Simultaneous mobility (c) Context-aware delivery

Figure 5: (a) Time-to-connect≈lookup latency for moderate mobility rates ($< \frac{1}{10s}$) as Auspice returns up-to-date responses w.h.p., but sharply rises thereafter (Eq. 2); (b) Simultaneous mobility recovery in ≈2 RTTs after both endpoints resurface; (c) Context-aware delivery showing 3 messages geo-cast to 5 members.

The *update propagation delay*, i.e., the time from when a client issued a write till the last replica executes the write, is a key determiner of the time-to-connect. As shown in §3.3.5, with eventual consistency, update propagation takes one round, while with total write ordering, update propagation takes two rounds and 50% more messages.

The measured update propagation delay is consistent with expectations. With eventual consistency, this delay is 154 ms, while with total write ordering, it is 292ms. Thus. the cost of the stronger consistency provided by total write ordering compared to eventual is that it can increase the time-to-connect latency by up to 2×. Note that the 2× inflation is a worst-case estimate, i.e., it will impact the time-to-connect latency only if a read request arrives at a replica while a write is under propagation to that replica, as we show below.

4.3 End-to-end mobility case studies

Can Auspice serve as the basis of a complete end-to-end mobility solution? To address this question, we have developed msocket, a user-level socket library that interoperates with Auspice, and supports all four types of endpoint mobility. The details of msocket's design and implementation is the subject of a separate paper [6]. Here, we *use* msocket to show proof-of-concept of some of Auspice's capabilities.

4.3.1 Time-to-connect to "moving" endpoints

We evaluate the time-to-connect to a moving destination as a function of the mobility (or update) rate. The *end-to-end time-to-connect* here is measured as the latency to look up an up-to-date address of the destination (or the time-to-connect as defined in §3.2) plus the time for msocket to successfully establish a TCP connection between the client and the mobile destination. This e2e-time-to-connect also incorporates the impact of timeouts and retried lookups if the client happens to have obtained a stale value (as in Fig. 1). The experiment is conducted on PlanetLab and consists of a single msocket client and a single mobile msocket server that is "moving" by changing its listening port number on a remote machine, and updating the name record replicated on three Auspice name servers accordingly. A successful connection setup delay using msocket is takes 2 RTTs (2 × 105 ms) [6]. As defined in Eq. 2, the values of the update propagation latency d_i and the lookup latency l_i are 250 ms and 20 ms respectively, and the update rate w_i varies from 1/1024/s to 1/s. The timeout value (T) in our experiment is dependent on the RTT between the client and the server. If the client attempts to connect to the server on a port which the server is not listening on, the server immediately returns an error response to the client. Specifically, the timeout value is either 1 or 2 RTTs with equal probability depending on whether the connection failed during the first or the second round-trip of msocket's connection setup. The client sends lookups at a rate of 10/s (but this rate does not affect the time-to-connect), and both lookups and updates inter-arrival times are exponentially distributed.

Figure 5(a) shows the distribution of the time-to-connect with update propagation delays entailed by eventual consistency. For low-to-moderate mobility rates ($< \frac{1}{64s}$), we find that all time-to-connect values are close to 230 ms, of which 20ms is the lookup latency, and 210ms is msocket's connection setup latency. The reason the client is able to obtain the correct value upon first lookup in all cases is that the update propagation latency of 250ms is much smaller than the average inter-update interval (64s). The update propagation delay becomes a non-trivial fraction of the inter-update interval at high mobility rates of ≈1/sec that results in 26% of lookups returning stale values. The mean e2e-time-to-connect increases to 302 ms for an update rate of 1/sec, which suggests that Auspice's time-to-connect is limited by network propagation delays in this regime. Nevertheless, once a connection is successfully established, individual migration can quickly resynchronize the connection in ≈two round-trips between the client and the mobile without relying on Auspice (not shown here).

Figure 5(a) also shows that the time-to-connect as predicted by our analytical model (Eq. 2) are close to those observed in the experiment, thereby re-affirming our design.

4.3.2 Simultaneous endpoint mobility

Figure 5(b) shows an experiment involving simultaneous mobility. The client is an Android phone using msocket via a WiFi interface to connect to a publicly addressable Planetlab machine at time 0. The server and client shut down their interfaces respectively around 15 and 20 sec. Subsequently, the server restarts its interface and starts listening on a different port and updates Auspice accordingly. After that, the client restarts its interface and attempts to re-synchronize the connection. This re-synchronization time is roughly 300ms as shown and consists of the following delays. The client performs a query to Auspice to resolve the

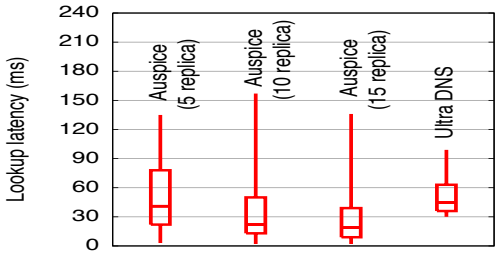


Figure 6: Lookup latency: Auspice with 5 replicas is comparable to UltraDNS (16 replicas); Auspice with 15 replicas has 60% lower latency than UltraDNS.

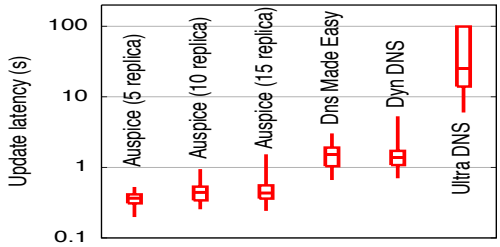


Figure 7: Update propagation delay: Auspice with 5 replicas is 1.0 to 24.7 secs lower than three top-tier managed DNS service providers.

server's GUID to its new socket address (IP, port), which takes roughly 50ms and mostly corresponds to the round-trip delay between the client and the Auspice nameserver. The remaining 250ms roughly correspond to 2 RTTs of delay between the client and the server that are separated by a round-trip delay of 120ms.

4.3.3 *Context-aware delivery*

Next, we show a proof of concept of context-aware communication, a novel communication primitive enabled by Auspice's extensible key-value API. Auspice allows applications to bind an msocket not only to human-readable names or GUIDs, but also to abstract context descriptors as in `msocket.bind("[geoloc: [lat,long],radius]")`. Writes to this msocket are reliably delivered to all GUIDs in the geo-fence created by this descriptor. Underneath the covers, msocket invokes Auspice to create on-demand a group GUID, i.e., a GUID with a membership field consisting of a set of member GUIDs, and obtains this member set. msocket internally resolves each member GUID to its socket address and establishes an msocket connection for reliably delivery.

Figure 5(c) shows an experiment involving a group creator (also the message sender) on an Android phone and a number of potential members on PlanetLab nodes, 5 of which fake-register their coordinates in Auspice so as to appear to be within the created geo-fence. The RTT between the group creator and members is 125ms. The figure shows that group creation, a single call to Auspice that returns all member GUIDs, takes roughly 200ms. Subsequently, an internal msocket connect to each member involves another Auspice lookup to resolve the GUID to a socket address and connect in parallel to all 5 members, which takes 250-280ms. After this, the creator sends 3 short messages back-to-back that each take roughly 1 RTT to be reliably delivered.

More details of optimizing context-aware queries in Auspice, reducing membership staleness, the connection migration protocol, etc. are outside the scope of this paper [6]. This experiment seeks only to exemplify a powerful, new communication primitive enabled by context descriptors compared to strictly hierarchical DNS names, as argued in §2.2.

4.4 Auspice vs. managed DNS providers

Can demand-aware replication benefit commercial managed DNS providers that largely rely on statically replicating today's (hardly mobile) domain names? To investigate this, we compare Auspice against three top-tier providers, UltraDNS, DynDNS, and DNSMadeEasy that offer geo-replicated authoritative DNS services widely used by enterprises (e.g., Dyn provides DNS service for Twitter).

4.4.1 *Lookup latency*

We compare Auspice to UltraDNS for a workload of lookups for domain names serviced by the provider. We identify 316 domain names among the top 10K Alexa websites serviced by this provider, and determine the geo-distribution of lookups for each name from their data [2]. For each name, we measure the latency for 1000 lookups from 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, `xqf4p.google.com` instead of `google.com`, that is unlikely to exist in a cache and requires an authoritative lookup. Auspice name servers are deployed across a total of 80 PlanetLab locations while UltraDNS has 16 known server locations [50]. We evaluate Auspice for three configurations with 5, 10, and 15 replicas of a name respectively.

Figure 6 shows the lookup latencies of names for Auspice and for UltraDNS. UltraDNS incurs a median latency of 45 ms with 16 replicas, while Auspice incurs 41 ms, 22 ms, and 18 ms respectively with 5, 10, and 15 replicas. With 5 replicas, Auspice's performance is comparable to UltraDNS with one-third the replication cost. With 15 replicas, Auspice incurs 60% lower latency for a comparable cost. The comparison against the other two, Dyn and DNSMadeEasy, is qualitatively similar [1]. Thus, Auspice's demand-aware replication achieves a better cost–performance tradeoff compared to static replication.

4.4.2 *Update propagation delay*

To measure update propagation delays, 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 immediately 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 propagation latency at a node is the time 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 (an upper bound on the update propagation delay). We show the distribution of measured update latencies for Auspice and for three managed DNS providers in Figure 7.

Auspice incurs lower update propagation latencies than all three providers for an equal or greater number of replica locations for names. We were unable to ascertain from UltraDNS why their update latencies are an order of magnitude higher than network propagation delays, but this finding is consistent with a recent study [50] that has shown latencies of up to tens of seconds for these providers. Indeed, some providers even distinguish themselves by advertising shorter update propagation delays than competitors [50].

Sensitivity analyses and other results.

We have conducted a comprehensive evaluation of the sensitivity of Auspice's performance-cost trade-offs to workload and system parameters across scales varying by several orders of magnitude. These include workload parameters such as geo-locality, read-to-write rate ratio, ratio of device-to-service names, etc. and system parameters such as the fault-tolerance threshold, capacity utilization, perturbation knob, the tunable overhead of replica reconfiguration, etc. using a combination of simulation and system experiments. These results do not qualitatively change the findings in this paper, and are deferred to the technical report [1].

5. RELATED WORK

Our work draws on lessons learned from an enormous body of prior work on network architecture as well as distributed systems, as described in §1 and §2.1. We discuss related work not covered elsewhere in the paper here.

DNS. Many have studied issues related to performance, scalability, load balancing, or denial-of-service vulnerabilities in DNS's resolution infrastructure [46, 49, 16, 22]. Several DHT-based alternatives have been put forward [49, 20, 45] and we compare against one representative proposal, Codons [49]. 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 a planned placement approach.

Vu et al. describe DMap [55], an in-network DHT scheme that is similar in spirit to Random-M as evaluated in our experiments (§4) (with a more direct comparison in [1]), showing that demand-aware placement can dramatically outperform randomized placement. A more important qualitative distinction is that DMap ties federation to the interconnection structure between ISPs, which entails commensurate lookup latency penalties and potential incentive mismatches by mapping GUIDs to non-provider ISPs. In comparison, the Auspice approach decouples the federation structure between GNS providers from that between ISPs.

Server selection. Many prior systems have addressed the server selection problem with data or services replicated across a wide-area network. Examples include anycast services [25, 15, 57] to map users to the best server based on server load or network path characteristics. These systems as well as CDNs and cloud hosting providers 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 balance.

Dynamic placement. We were unable to find prior systems that *automatically* reconfigure the *geo-distributed* replica locations of frequently *mutable objects* while preserving *consistency* (i.e., those satisfying all four italicized properties). However, reconfigurable placement has been studied for static or slow changing content [30] or within a single datacenter, or without replication. For example, Volley [11] optimizes the placement of mutable data objects based on the geo-distribution of accesses and is similar in spirit to Auspice in this respect, however it implicitly assumes a single replica for each object, so it does not have to worry about high update rates or replica coordination overhead.

Auspice is related to many distributed key-value stores [5, 23, 3], most of which are optimized for distribution within, not across, data centers. Some (e.g., Cassandra) support a geo-distributed deployment using a fixed number of replica sites. Spanner [19] is a geo-distributed data store that synchronously replicates data ("directories") across datacenters with a semi-relational database abstraction. Compared to Spanner, Auspice does not provide any guarantees on operations spanning multiple records, but unlike Spanner's geographic placement of replicas that "administrators control" by creating a "menu of named options", Auspice automatically reconfigures the number and placement of replicas so as to reduce lookup latency and update cost. Furthermore, Spanner assigns a large number of directory objects to a much smaller number of fixed Paxos groups; Auspice supports an arbitrarily reconfigurable Paxos group per object based on principles in recent theoretical work on reconfigurable consensus, e.g., Vertical Paxos [39] and the more recent report on Viewstamped Replication Revisited [41].

6. CONCLUSIONS

In this paper, we presented the design, implementation, and evaluation of Auspice, a scalable, geo-distributed, federated global name service for any Internetwork 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 strategy can significantly improve the performance-cost tradeoffs struck both by commercial managed DNS services employing simplistic replication strategies today as well as previously proposed DHT-based replication alternatives with or without high mobility. Our case studies confirm that Auspice can form the basis of an end-to-end mobility solution and also enable novel context-aware communication primitives that generalize name- or address-based communication. A pre-release version of Auspice on EC2 can be accessed through the developer portal at `http://gns.name`.

Acknowledgments. This research was funded in part by CNS-1040781 and CNS-0845855. We thank the rest of the MobilityFirst team, the paper's past and recent reviewers, our shepherd Ellen Zegura, Anand Seetharam, Emmanuel Cecchet, Jim Kurose, Marvin Sirbu, and NSF-FIA meeting participants for their feedback.

7. REFERENCES

[1] A Global Name Service for a Highly Mobile Internetwork. UMass SCS Technical Report, 2013 and 2014. https://web.cs.umass.edu/publication.

[2] Alexa Web Information Service. http://www.alexa.com.

[3] Cassandra. http://cassandra.apache.org.

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

[5] mongoDB. http://www.mongodb.org/.

[6] msocket: System Support for Developing Seamlessly Mobile, Multipath, and Middlebox-Agnostic Applications. UMass SCS Technical Report, 2014. https://web.cs.umass.edu/publication.

[7] Server fault: DNS - Any way to force a name server to update the record of a domain? http://serverfault.com/questions/41018.

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

[9] ICANN Hears Concerns about Accountability, Control, October 2008. http://www.infoworld.com/t/networking/icann-hears-concerns-about-accountability-control-216.

[10] Debate Rages over who Should Control ICANN. *Processor*, 31(16), June 2009.

[11] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *USENIX NSDI*, 2010.

[12] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol. In *ACM SIGCOMM*, 2008.

[13] 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.

[14] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *ACM SIGCOMM*, 2004.

[15] S. Bhattacharjee and et al. Application-Layer Anycasting. In *IEEE INFOCOM*, 1997.

[16] N. Brownlee, K. Claffy, and E. Nemeth. DNS Measurements at a Root Server. In *GLOBECOM*, 2001.

[17] M. Caesar, T. Condie, and J. Kannan et al. ROFL: Routing on Flat Labels. In *ACM SIGCOMM*, 2006.

[18] Cisco. Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012-2017. http://ciscovni.com.

[19] J. C. Corbett and J. Dean et al. Spanner: Google's Globally Distributed Database. *USENIX OSDI*, 2012.

[20] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS Using a Peer-to-Peer Lookup Service. In *IPTPS*, 2002.

[21] G. DeCandia and et al. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SOSP*, 2007.

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

[23] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM SIGCOMM*, 2012.

[24] A. Feldmann and et al. HAIR: Hierarchical Architecture for Internet Routing. In *ReArch Workshop*, 2009.

[25] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for Any Service. In *USENIX NSDI*, 2006.

[26] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. In *ICNP*, 1997.

[27] Z. Gao, A. Venkataramani, and J. F. Kurose. Towards a quantitative comparison of location-independent network architectures. In *ACM SIGCOMM*, 2014.

[28] Gartner. Sales of Android Phones to Approach One Billion in 2014. http://www.gartner.com/newsroom/id/2665715.

[29] M. Gritter and D. R. Cheriton. An Architecture for Content Routing Support in the Internet. In *USITS*, 2001.

[30] J. Gwertzman and M. Seltzer. The case for geographical push caching. In *IEEE HotOS Workshop*, May 1995.

[31] D. Han, A. Anand, F. Dogar et al., B. Li, H. Lim, and M. et al. XIA: Efficient Support for Evolvable Internetworking. In *USENIX NSDI*, 2012.

[32] V. Jacobson and et al. Networking Named Content. In *ACM SIGCOMM CoNEXT*, 2009.

[33] P. Jokela, P. Nikander, J. Melen, J. Ylitalo, and J. Wall. Host Identity Protocol, Extended Abstract. In *Wireless World Research Forum*, 2004.

[34] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on Networking*, October 2002.

[35] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A Data-Oriented (and Beyond) Network Architecture. In *ACM SIGCOMM*, 2007.

[36] D. Krioukov and et al. On Compact Routing for the Internet. *ACM SIGCOMM CCR*, 2007.

[37] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *WWW*, 2010.

[38] L. Lamport. The Part-Time Parliament. *ACM Transactions on Compututer Systems*, 16(2):133–169, May 1998.

[39] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-Backup Replication. In *ACM PODC*, 2009.

[40] B. W. Lampson. Designing a Global Name Service. In *ACM PODC*, 1986.

[41] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT CSAIL-TR-2012-021, 2012.

[42] H. V. Madhyastha and et al. iPlane: An Information Plane for Distributed Services. In *USNIX OSDI*, 2006.

[43] P. Mockapetris and K. J. Dunlap. Development of the domain name system. ACM SIGCOMM, 1988.

[44] E. Nordstrom and et al. Serval: An End-Host Stack for Service-Centric Networking. In *USENIX NSDI*, 2012.

[45] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A Comparative Study of the DNS Design with DHT-Based Alternatives. In *IEEE INFOCOM*, 2006.

[46] V. Pappas, Z. Xu, S. Lu, D. Massey, A. Terzis, and L. Zhang. Impact of Configuration Errors on DNS Robustness. In *ACM SIGCOMM*, 2004.

[47] M. Parwez and et al. DNS propagation delay: An effective and robust solution using authoritative response from non-authoritative server. In *ICIME*, 2010.

[48] C. Perkins. RFC 3220: IP Mobility Support for IPv4, 2002.

[49] V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *ACM SIGCOMM*, 2004.

[50] J. Read. Comparison and Analysis of Managed DNS Providers, Aug 2012. Cloud Harmony Inc.

[51] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with Grapevine: the Growth of a Distributed System. *ACM Trans. Comput. Syst.*, 1984.

[52] A. C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *ACM MobiCom*, 2000.

[53] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *SIGCOMM*, 2002.

[54] A. Venkataramani, J. Kurose, D. Raychaudhuri, K. Nagaraja, M. Mao, and S. Banerjee. MobilityFirst: A Mobility-Centric and Trustworthy Internet Architecture. *ACM SIGCOMM CCR*, 2014.

[55] T. Vu and et al. DMap: A Shared Hosting Scheme for Dynamic Identifier to Locator Mappings in the Global Internet. In *IEEE ICDCS*, 2012.

[56] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *OSDI*, 2004.

[57] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford. DONAR: Decentralized Server Selection for Cloud Services. In *ACM SIGCOMM*, 2010.