CHAPTER 6

DEMAND-AWARE GEO-DISTRIBUTED PLACEMENT FOR LOW LATENCY

Geo-distributed computing clouds such as Amazon EC2, Google offer a scalable, fault tolerant and flexible solution for hosting web services that cater to users world-wide. The easy availability of these platforms provides opportunities to several applications to leverage the performance benefits of geo-distribution.

A challenge for a web service in switching to a geo-distributed cloud is the cost of replication of the "data-tier". Several web services today generate dynamic content such as weather information, stock prices, and status updates posted by users on a social networking website. Data replication is necessary to reduce latency of content accesses but is a costly operation for dynamic content. The reason is that the cost of propagating updates to dynamic content increases linearly with the number of locations. State-of-art replication alternatives either provide poor cost-vs-performance tradeoffs or leave data placement to be done manually by web-services which increases human cost and effort. For example, DHT-based designs make a constant number of replicas but result in high latency of content accesses.

Our key insight is that replication of dynamic data should be done in a demand-aware manner such that a limited number of data replicas placed close to pockets of demand are sufficient to reduce latency of content accesses. A demand-aware placement implicitly assumes geo-locality of workloads, an assumption we believe is justified based on recent studies of workload characteristics. We have developed a heuristic placement strategy that decides number of replicas based on read-to-write ratio of a name and selects replica locations based on geo-distribution of demand to provide low lookup latency, low update cost, and high availability.

Our system, Auspice, is implemented as a geo-distributed key-value store that stores arbitrary JSON objects as records. Like several other key-value stores [8, 47, 5], Auspice exposes a simple GET/PUT interface to clients. Auspice provides flexible consistency semantics for accesses to a single object. However, Auspice is not a general-purpose database and lacks several features that are common in a database, e.g., support for running SELECT queries efficiently, or support for transactions. Auspice is scalable to a large number of locations and data items due to a fully decentralized design both in the data plane and and the control plane that makes data placement decisions.

An application suited for Auspice is a global name service that provide name-to-address mapping for mobile devices. We have evaluated Auspice extensively for an expected workload of such a global name service. Our experiments show that Auspice provides 5.4×-11.2× lower lookup latency than a DHT-based design for DNS. In a comparison to commercial managed DNS providers, we find that Auspice provides a median update latency that is 1.1 sec to 24.7 sec lower than three top-tier providers.

6.1 Auspice design & implementation

6.1.1 Design goals

Our goal is to design a distributed system that meets the following requirements:

(1) Low lookup latency: Replicas of a key should be placed close to end-users accessing it so as to minimize user-perceived response times.

(2) Resource cost: The design must ensure low replication cost. A naive way to minimize lookup latencies is to replicate every 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) Consistency: The design must provide flexible consistency semantics as desired by an application.

6.1.2 Auspice's geo-distributed design

To address the above goals, the Auspice 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 is chosen for its popularity among today's web services. Each *record* in Auspice is associated with a globally unique identifier (GUID) that is the record's primary key. A 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 $\mid K_1, V_1 \mid K_2, V_2 \mid \cdots$

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 record in accordance with (1) the lookup and update request rates for the record, (2) the geo-distribution of requests for the record, and (3) the aggregate request load across all records.

Figure 6.1 illustrates the placement engine. Each record is associated with a fixed number, F, of *replica-controllers* and a variable number of *active replicas* of the corresponding record. The record'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 records and nodes. The replica-controllers

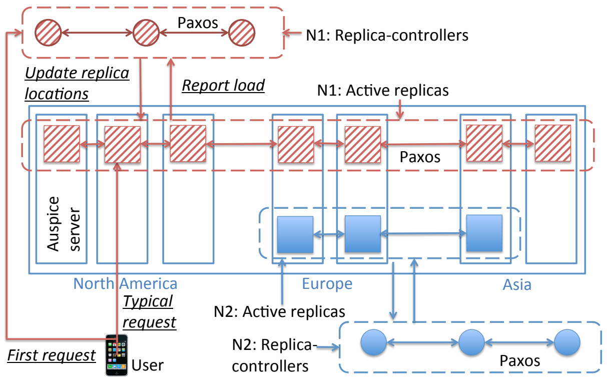


Figure 6.1. Geo-distributed servers in Auspice. Replica-controllers (logically separate from active replicas) decide placement of active replicas and active replicas handle requests from end-users. N1 is a globally popular record and is replicated globally; record N2 is popular in select regions and is replicated in those regions.

are responsible only for determining the number and locations of the active replicas, and the actives replicas are responsible for maintaining the actual record and processing client requests. The replica-controllers implement a replicated state machine using Paxos [72] in order to maintain a consistent view of the current set of active replicas.

A record'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 record 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 record.

6.1.2.1 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 servers to determine the number and locations of active replicas for each record 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 record 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 record while placing the rest randomly so as to balance the potentially conflicting goals of reducing latency and balancing load among servers.

Specifically, the placement algorithm computes the number of replicas for a record as $(F + \beta r_i / w_i)$, where r_i and w_i are the lookup and update rates of record i; F is the minimum number of replicas needed to meet the availability goal (§6.1.1); 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 servers (in reads/sec) is C. Then, β is set so that

$$\mu C = \sum_i r_i + \sum_i (F + \beta \frac{r_i}{w_i}) w_i \tag{6.1}$$

where the right hand side represents the total load summed across all records. 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 record 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 $\nu(F + \beta r_i / w_i)$ servers that are respectively the closest to the top $\nu(F + \beta r_i / w_i)$ regions sorted by demand for record 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 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.

Auspice's replica placement scheme (Eq. 6.1) 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 record, where M is the total number of server locations. Thus, at light load, Auspice may replicate every record at every location, while under heavy load, it may create exactly F replicas for all but the most popular records.

6.1.2.2 Client request routing

A client request is routed from an end-host to a suitable server as follows. The set of all servers in an Auspice instance is known to each member server and can be obtained from a well-known location. When a client encounters a request for a record for the first time, it uses the known set of all servers and consistent hashing to determine the replica-controllers for that record and sends the request to the closest replica-controller. The replica-controller returns the set of active replicas for the record 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 client. Each client maintains an estimate of the round-trip latency to all 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 [81]. To incorporate load-induced latency, the latency estimate to a server is passively measured as a moving average over lookups sent to that server. The client also maintains a timeout value based on the moving average and variance of the estimates. If a lookup request sent to a server times out, the client infers that either the server or network route is congested, and it multiplicatively increases its latency estimate to that server by a fixed factor. Thus, if multiple lookups sent to a server time out, the estimated latency shoots up and the client stops sending requests to that 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).

6.1.2.3 Consistency with static replication

As a geo-distributed key-value store, Auspice must at least ensure this eventual consistency property: *all active replicas must eventually return the same value of the 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 record and no replica failures for sufficiently long. Violating this property means that a client may be indefinitely unable to obtain the the up-to-date value of the record even though the record is no longer being updated.

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 [39], 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 ordering. To be useful to a broader set of applications that demand more sophisticated query patterns, it may be useful in some scenarios to ensure that update operations (like appending to or deleting from a list) to a record are applied in the same order by all active replicas. Ensuring a total ordering of all updates to a record is a stronger property than eventual consistency, calling for a state-machine approach, which Auspice supports as an option. In fact, Auspice supports an option to perform total ordering of all updates and lookups as well, which is a even stronger property than total write ordering alone.

To this end, active replicas for a record participate in a Paxos instance maintained separately for each record (distinct from Paxos used by replica-controllers to compute active replicas for that record). 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).

6.1.2.4 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 record to its *current* set of active replicas as obtained from any of the consistently-hashed replica-controllers for the record. 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 [73], 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].

6.1.2.5 Replica reconfiguration policies

The policy decision of when to reconfigure the replica group for a specific record is orthogonal to the consistency mechanisms above. The frequency of reconfiguration presents a tradeoff between agility to demand and the reconfiguration messaging overhead. This overhead includes (1) load reports per epoch from active replicas to replica-controllers; (2) replica-controllers computing the new set of active replicas by consensus; (3) replica-controllers notifying the old active replicas to stop and the new ones to start; and (4) newly added replicas obtaining the record state from any old replica, and (5) replica-controllers agreeing that the group change is complete. Each of these steps entails one or two small messages per record at each active replica or replica controller, except for step 4 where the overhead depends on the size of the record. The first overhead is incurred per epoch and can be further reduced by having an active replica issue a load report for a record only if the geo-locality of demand for the record has changed beyond a threshold at that replica. The subsequent steps are needed only if the policy deems reconfiguration as warranted based on the record's aggregate load report (noting that only a subset of records may need reconfiguration in an epoch).

Thus, any reconfiguration policy that (1) limits load reports at an active replica to at most once per m client requests for the record at that replica, and (2) limits reconfiguration at replica-controllers to at most once per km total client requests across the k currently active replicas, suffices to ensure that the reconfiguration overhead is at most a fraction $\approx 1/m$ of the incoming client request load. As a consequence, it may take at least m requests for a record at a new location before an active replica for it is created there.

6.1.2.6 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 servers. We do not rely on any distributed deployment features therein as the coordination middleware is what Auspice provides.

6.2 Evaluation

Our evaluation is centered around an representative application: a global name service (GNS) to provide name-to-address mapping for mobile devices. We expect a GNS to receive requests from clients spread in a large geographic area and the records in GNS to show a non-trivial update rate due to end-host mobility. These traits make a GNS a representative application for Auspice. In the following discussion, we refer an Auspice server as a *name server*, an Auspice client as a *local name server*, and a record as a *name record* or a *name*.

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) How does Auspice's cost-performance tradeoff compare to best-of-breed managed DNS services for today's (hardly mobile) domain name workloads?

6.2.1 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

Workload parameter	Value
Fraction of (highly mobile) device names	90%
Fraction of (mostly static) service names	10%
% 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: [devices, services]	[0.75, 0.8]

Table 6.1. Default workload parameters.

devices do not have publicly visible IP addresses. So we conduct an evaluation using synthetic workloads for device names (§6.2.2), 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 [71], 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 [3]. 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).

Table 6.1 summarizes the default workload parameters.

Replication schemes compared: Auspice uses the replica placement strategy as described in §6.1 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[98]. 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 Auspice for a fair comparison. All schemes direct a lookup to the closest available replica after the first request.

6.2.2 Evaluating Auspice's replica placement

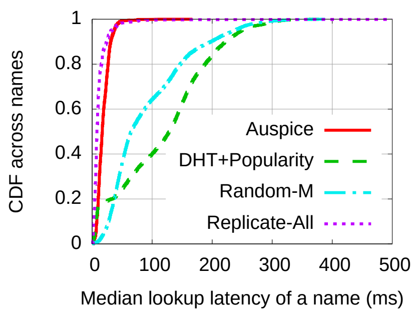
We conduct experiments in this subsection on a 16-node (each with Xeon 5140, 4-cores, 8 GB RAM) departmental cluster, wherein each machine hosts 10 instances of either name servers or local name servers so as to emulate an 80-name server Auspice deployment. We instrument the instances so as to emulate 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.

6.2.2.1 Lookup latency and update cost

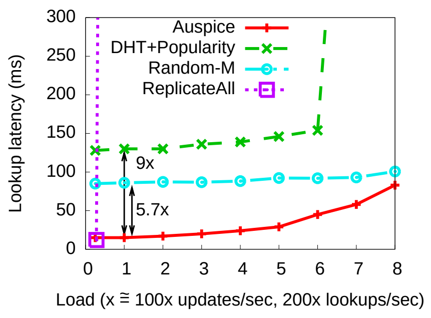
How well does Auspice use available resources for replicating name records? To evaluate this, we compare the lookup latency of schemes across varying load levels. A machine running 10 name servers receives on average 2000 lookups/sec and 1000 updates/sec at a load = 1. For each scheme, load is increased 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 6.2(a) shows the distribution of median lookup latency across names at the smallest load level (load = 0.3). Figure 6.2(b) shows load-vs-lookup latency curve for schemes, where "lookup latency" refers to the mean of the median lookup latencies of names. Figure 6.2(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.

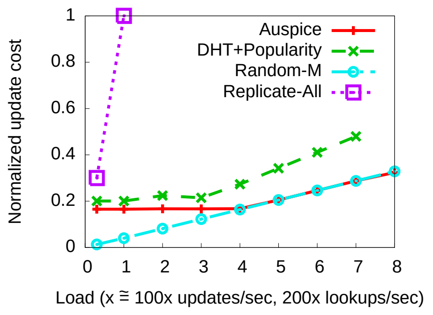
Replicate-All gives low lookup latencies at the smallest load level, but generates a very high update cost and can sustain a request load of at most 0.3. This is further supported by Figure 6.2(c) that shows that the update cost for Replicate-All at load = 0.4 is more than the update cost of Auspice 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. 6.2(b)) due to its low update costs, but its lookup latencies are higher as it only creates 3 replicas randomly.



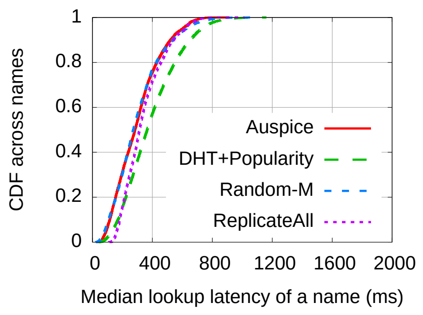
(a) Lookup latencies (load = 0.3)



(b) Lookup latency vs. load



(c) Update cost vs. load



(d) Update latency

Figure 6.2. Auspice has up to 5.7× to 9× lower latencies than Random-M and DHT+Popularity reps. (6.2(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. In Figure 6.2(d), median update latency of Auspice with total write ordering per name is 284ms and is comparable to other schemes.

Auspice has $5.7\times - 9\times$ lower latencies over Random-M and DHT+Popularity respectively (Figure 6.2(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. 6.1), thereby achieving low latencies for loads$\leq$4. At loads $\geq$ 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 $\geq$ 4 (Figure 6.2(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.

6.2.2.2 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 §6.2.2.1 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.

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. 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 $2\times$. Note that the $2\times$ 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.

6.2.2.3 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. This experiment is performed with the help of msocket [9], a user-level socket library that interoperates with Auspice. 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 §6.1.2) 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.

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) [9]. 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 6.3 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 $\approx$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 $\approx$two round-trips between the client and the mobile without relying on Auspice (not shown here).

We have also developed an analytical model to predict time-to-connect values [1]. Figure 6.3 also shows that the time-to-connect as predicted by our analytical model are close to those observed in the experiment, thereby re-affirming our design.

6.2.2.4 Reconfiguration overhead vs. responsiveness

In the next experiment, we show how Auspice can choose the epoch length and reconfiguration trigger so as to limit the overhead of reconfiguration (as in §6.1.2.5) while being responsive to changes in demand geo-locality. The workload changes the geo-distribution of demand for each name every 300 sec. For each name, we change the regions from which most requests arise so that changing the placement of replicas becomes necessary to minimize lookup latencies. The experiment is performed on a local cluster with a workload of 1000 names with characteristics as described in Table 6.1. The epoch length of group changes is chosen to be 75 sec, which ensures that group changes result in less than 10% overhead to the system (§6.1.2.5).

We show the lookup latencies of Auspice and the message overhead of reconfiguration in the experiment in Figue 6.4. We find that Auspice takes two epochs to infer a change in the geo-distribution of demand and to adapt to it. This result

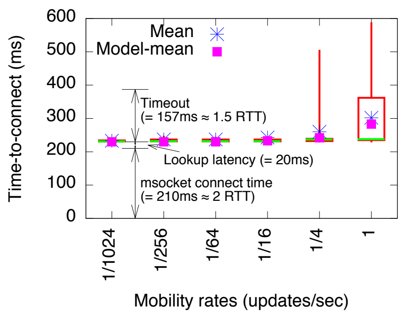


Figure 6.3. Time-to-connect ≈ lookup latency for moderate mobility rates (<1/100s) as Auspice returns up-to-date responses w.h.p., but sharply rises thereafter.

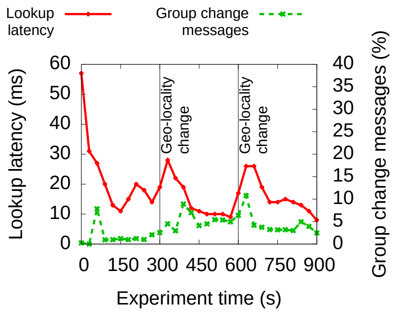


Figure 6.4. Auspice adapts to varying demand geo-locality in two epochs each 75s long with a (tunable) reconfiguration message overhead of 3.6%.

suggests that Auspice can optimize lookup latencies provided the geo-distribution of demand for a name remains stable for a few epochs. Further, we measured the overhead of reconfiguration messages in this experiment to be 3.6%. The overhead is less than the expected overhead of 10% because not all names are reconfigured in every epoch.

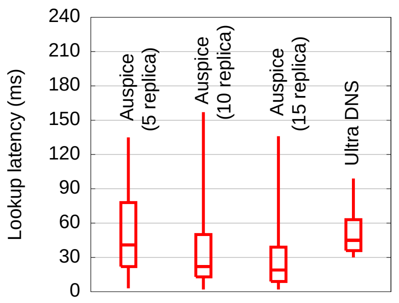


Figure 6.5. Lookup latency: Auspice with 5 replicas is comparable to UltraDNS (16 replicas); Auspice with 15 replicas has 60% lower latency than UltraDNS.

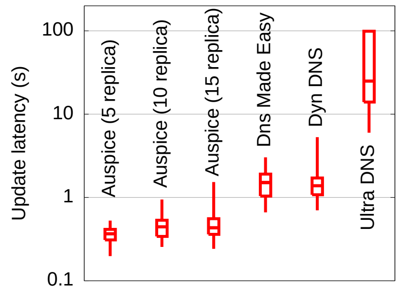


Figure 6.6. Update propagation delay: Auspice with 5 replicas is 1.0 to 24.7 secs lower than three top-tier managed DNS service providers.

6.2.3 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).

6.2.3.1 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 [3]. 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 [101]. We evaluate Auspice for three configurations with 5, 10, and 15 replicas of a name respectively.

Figure 6.5 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.

6.2.3.2 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 6.6.

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 [101] 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 [101].

6.2.4 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 above findings, and are deferred to the technical report [1].

6.3 Related work

Our work draws on lessons learned from an enormous body of prior work on distributed systems as discussed below.

DNS. Many have studied issues related to performance, scalability, load balancing, or denial-of-service vulnerabilities in DNS's resolution infrastructure [91, 98, 28, 44]. Several DHT-based alternatives have been put forward [98, 37, 90] and we compare against one representative proposal, Codons [98]. 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 [116], an in-network DHT scheme that is similar in spirit to Random-M as evaluated in our experiments (§6.2) (with a more direct comparison in [1]), showing that demand-aware placement can dramatically outperform randomized placement.

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 [56, 26, 118] 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 [62] or within a single datacenter, or without replication. For example, Volley [14] 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 [8, 47, 5], 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 [36] 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 [73] and the more recent report on Viewstamped Replication Revisited [77].

6.4 Conclusions

We presented the design, implementation, and evaluation of Auspice, a scalable, geo-distributed, key-value store. At the core of Auspice is a placement engine for replicating records to achieve low lookup latency, low update cost, and high availability. We have extensively evaluated Auspice for a representative application: a global name service to provide name-to-address mapping for mobile devices. 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.