DONAR: Decentralized Server Selection for Cloud Services

Patrick Wendell, Joe Wenjie Jiang, Michael J. Freedman, and Jennifer Rexford
Department of Computer Science, Princeton University

ABSTRACT

Geo-replicated services need an effective way to direct client requests to a particular location, based on performance, load, and cost. This paper presents DONAR, a distributed system that can offload the burden of replica selection, while providing these services with a sufficiently expressive interface for specifying mapping policies. Most existing approaches for replica selection rely on either central coordination (which has reliability, security, and scalability limitations) or distributed heuristics (which lead to suboptimal request distributions, or even instability). In contrast, the distributed mapping nodes in DONAR run a simple, efficient algorithm to coordinate their replica-selection decisions for clients. The protocol solves an optimization problem that jointly considers both client performance and server load, allowing us to show that the distributed algorithm is stable and effective. Experiments with our DONAR prototype—providing replica selection for CoralCDN and the Measurement Lab—demonstrate that our algorithm performs well "in the wild." Our prototype supports DNS- and HTTP-based redirection, IP anycast, and a secure update protocol, and can handle many customer services with diverse policy objectives.

Categories and Subject Descriptors: H.3.4 [Information Systems]: Systems and Software—*Distributed Systems*; C.2.4 [Computer – Communication Networks]: Distributed Systems—*Distributed Applications*

General Terms: Algorithms, Design

Keywords: Replica Selection, Load Balancing, Geo-Locality, DNS, Distributed Optimization

1. INTRODUCTION

The Internet is increasingly a platform for online services—such as Web search, social networks, and video streaming—distributed across multiple locations for better reliability and performance. The trend toward geographically-diverse server placement will only continue and increasingly include smaller enterprises, with the success of cloud-computing platforms like Amazon AWS [1]. These services all need an effective way to direct clients across the wide area to an appropriate service location (or "replica"). For many companies offering distributed services, managing replica selection is an unnecessary burden. In this paper, we present the design, implementation, evaluation, and deployment of DONAR, a decentralized replica-selection system that meets the needs of these services.

SIGCOMM'10, August 30–September 3, 2010, New Delhi, India.

1.1 A Case for Outsourcing Replica Selection

Many networked services handle replica selection themselves. However, even the simplest approach of using DNS-based replica selection requires running a DNS server that tracks changes in which replicas are running and customizes the IP address(es) returned to different clients. These IP addresses may represent single servers in some cases. Or, they may be virtualized addresses that each represent a cluster of collocated machines, with an on-path load balancer directing requests to individual servers. To handle wide-area replica selection *well*, these companies need to (i) run the DNS server at multiple locations, for better reliability and performance, (ii) have these nodes coordinate to distribute client requests across the replicas, to strike a good trade-off between client performance, server load, and network cost, and (iii) perhaps switch from DNS to alternate techniques, like HTTP-based redirection or proxying, that offer finer-grain control over directing requests to replicas.

One alternative is to outsource the entire responsibility for running a Web-based service to a CDN with ample server and network capacity (*e.g.*, Akamai [2]). Increasingly, cloud computing offers an attractive alternative where the cloud provider offers elastic server and network resources, while allowing customers to design and implement their own services. Today, such customers are left largely to handle the replica-selection process on their own, with at best limited support from individual cloud providers [3] and third-party DNS hosting platforms [4, 5].

Instead, companies should be able to manage their own distributed services while "outsourcing" replica selection to a third party or their cloud provider(s). These companies should merely specify high-level *policies*, based on performance, server and network load, and cost. Then, the replica-selection system should realize these policies by directing clients to the appropriate replicas and adapting to policy changes, server replica failures, and shifts in the client demands. To be effective, the replica-selection system must satisfy several important goals for its customers. It must be:

- **Expressive:** Customers should have a sufficiently expressive interface to specify policies based on (some combination of) performance, replica load, and server and bandwidth costs.
- **Reliable:** The system should offer reliable service to clients, as well as stable storage of customer policy and replica configuration data.
- **Accurate:** Client requests should be directed to the service replicas as accurately as possible, based on the customer's replica-selection policy.

- **Responsive:** The replica-selection system should respond quickly to changing client demands and customer policies without introducing instability.
- **Flexible:** The nodes should support a variety of replica-selection mechanisms (*e.g.*, DNS and HTTP-redirection).
- **Secure:** Only the customer, or another authorized party, should be able to create or change its selection policies.

In this paper, we present the design, implementation, evaluation, and deployment of DONAR, a decentralized replica-selection system that achieves these goals. DONAR's distributed algorithm solves a formal optimization problem that jointly considers both client locality, server load, and policy preferences. By design, DONAR facilitates replica selection for *many* services, however its underlying algorithms remain relevant in the case of a single service performing its own replica selection, such as a commercial CDN.

1.2 Decentralized Replica-Selection System

The need for reliability and performance should drive the design of a replica-selection system, leading to a *distributed* solution that consists of multiple *mapping nodes* handling a diverse mix of clients, as shown in Figure 1. These mapping nodes could be HTTP ingress proxies that route client requests from a given locale to the appropriate data centers, the model adopted by Google and Yahoo. Or the mapping nodes could be authoritative DNS servers that resolve local queries for the names of Web sites, the model adopted by Akamai and most CDNs. Furthermore, these DNS servers may use IP anycast to leverage BGP-based failover and to minimize client request latency. Whatever the mechanism for interacting with clients, each mapping nodes has only a partial view of the global space of clients. As such, these mapping nodes need to make different decisions; for example, node 1 in Figure 1 directs all of its clients to the leftmost replica, whereas node 3 divides its clients among the remaining three replicas.

Each mapping node needs to know how to both direct its clients and adapt to changing conditions. The simplest approach would be to have a central coordinator that collects information about the mix of clients per customer service, as well as the request load from each mapping node, and then informs each mapping node how to direct future requests. However, a central coordinator introduces a single point of failure, as well as an attractive target for attackers trying to bring down the service. Further, it incurs significant overhead for the mapping nodes to interact with the controller. While some existing services do perform centralized computation—for example, Akamai uses a centralized hierarchical stable-marriage algorithm for assigning clients to its CDN servers [6]—the overhead of backhauling client information can be more prohibitive when it must be done for *each* customer service using DONAR. Finally, a centralized solution adds additional delay, making the system less responsive to sudden changes in client request rates (*i.e.*, flash crowds).

To overcome these limitations, DONAR runs a *decentralized* algorithm among the mapping nodes themselves, with minimal protocol overhead that does not grow with the number of clients. Designing a distributed algorithm that is simultaneously scalable, accurate, and responsive is an open challenge, not addressed by previous heuristic-based [7, 8, 9, 10] or partially centralized [11] solutions. Decentralized algorithms are notoriously prone to *oscillations* (where the nodes over-react based on their own local information) and *inaccuracy* (where the system does not balance replica load effectively). DONAR must avoid falling into these traps.

1.3 Research Contributions and Roadmap

In designing, implementing, deploying, and evaluating DONAR, we make three main research contributions:

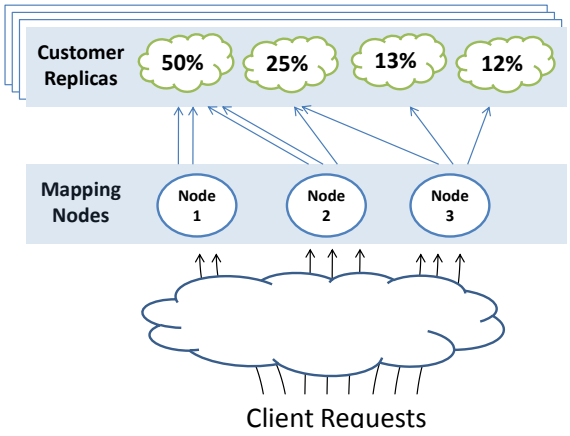

Figure 1: DONAR uses distributed mapping nodes for replica selection. Its algorithms can maintain a weighted split of requests to a customer's replicas, while preserving client–replica locality to the greatest extent possible.

Simple and expressive interface for customer policies (Section 2): DONAR has a simple policy interface where each replica location specifies a *split weight* or a *bandwidth cap*, and expected client performance is captured through a *performance penalty*. These three sets of parameters may change over time. We show that this interface can capture diverse replica-selection goals based on client proximity, server capacity, 95th-percentile billing, and so on. The API leads us to introduce a formal optimization problem that jointly consider customers' (sometimes conflicting) preferences.

Stable, efficient, and accurate distributed replica-selection algorithm (Section 3): DONAR consists of a distributed collection of mapping nodes for better reliability and performance. Since these nodes each handle replica selection for a different mix of clients, they cannot simply solve the optimization problem independently. Rather than resorting to a centralized architecture that pushes results to each mapping node, we decompose the optimization problem into a distributed algorithm that requires only minimal coordination between the nodes. We show that our decentralized algorithm provably converges to the solution of the optimization problem, ensuring that DONAR does not over or under react to shifts in client demands.

Scalable, secure, reliable, and flexible prototype system (Section 4): Our DONAR prototype implements the distributed optimization algorithm, supports DNS- and HTTP-based replica selection, and stores customer data in a scalable storage system. DONAR uses IP anycast for fast failover and good client redirection performance. The prototype includes a secure update protocol for policy changes and receives a periodic feed of IP2Geo data [12] to support policies based on client location. Our prototype is used to provide distributed DNS resolution for the Measurement Lab testbed [13] and for a portion of the CoralCDN service [14].

Experiments in Section 5 evaluate both our distributed algorithm operating at scale (through trace-driven simulations of client requests to CoralCDN) and a small-scale deployment of our prototype system (providing DNS-based replica selection to jointly optimize client proximity and server load for part of CoralCDN). These experiments demonstrate that DONAR offers effective, customized replica selection in a scalable and efficient fashion. Section 6 compares DONAR to related work, and Section 7 concludes.

2. CONFIGURABLE MAPPING POLICIES

DONAR realizes its customers' high-level policy goals, while shielding them from the complicated internals of distributed replica

selection. By *customer*, we mean a service provider that outsources replica selection to DONAR. Customers configure their policies by communicating directly with any DONAR mapping node. This section first motivates and introduces DONAR's policy API. It then describes a number of application scenarios that leverage this interface to express sophisticated mapping policies.

2.1 Customer Goals

Customers use DONAR to optimally pair clients with service replicas. What customers consider "optimal" can differ: some may seek simply to minimize the network latency between clients and replicas, others may seek to balance load across all replicas, while still others may try to optimize the assignment based on the billing costs of their network operators or hosting services. In general, however, we can decompose a customer's preferences into those associated with the *wide-area network* (the network performance a client would experience if directed to a particular replica) and those associated with *its own replicas* (the load on the servers and the network at each location). DONAR considers *both* factors in its replica-selection algorithm.

Minimizing network costs. Mapping policies commonly seek to pair clients with replicas that offer good performance. While replica load can affect client-perceived performance, the network path has a significant impact as well. For example, web transfers or interactive applications often seek small network round-trip times (RTTs), while bulk transfers seek good network throughput (although RTT certainly can also impact TCP throughput).

DONAR's algorithms use an abstract $cost(c,i)$ function to indicate the performance penalty between a particular client–replica pair. This function allows us a considerable amount of expressiveness. For instance, to optimize latency, *cost* simply can be RTT, directly measured or estimated via network coordinates. If throughput is the primary goal, *cost* can be calculated as the penalty of network congestion. In fact, the *cost* function can be any shape—*e.g.*, a translated logistic function of latency or congestion—to optimize the worst case or percentile-based performance. The flexibility of the *cost* function also allows for intricate policies, *e.g.*, always mapping one client to a particular server replica, or preferring a replica through a peering link over a transit link.

DONAR's current implementation starts with a shared *cost* function for all of its customers, which represents expected path latency. As a result, our customers do not need to calculate and submit large and complex cost functions independently. In the case where a customer's notion of cost differs from that shared function, our interface allows them to override DONAR's default mapping decisions (discussed in more detail in the following section).

To estimate path latency, services like DONAR could use a variety of techniques: direct network measurements [15, 2, 16], virtual coordinates [17, 18], structural models of path performance [19, 20], or some hybrid of these approaches [10]. DONAR's algorithm can accept any of these techniques; research towards improving network cost estimation is very complementary to our interests. Our current prototype uses a commercial IP geolocation database [12] to estimate network proximity, although this easily could be replaced with an alternative solution.

Balancing client requests across replicas. Unlike pairwise network performance, which is largely shared amongst DONAR's customers, traffic distribution preferences vary widely from customer to customer. In the simplest scenarios, services may want to equally balance request rates across replicas, or they may want decisions based solely on network proximity (*i.e.*, *cost*). More advanced considerations, however, are both possible and important. For example, given 95th-percentile billing mechanisms, customers could try to minimize costs by reducing the frequency of peak consumption, or to eliminate overage costs by rarely exceeding their committed rates. In any replica-mapping scheme, request capacity, bandwidth cost, and other factors will influence the preferred rate of request arrival at a given replica. The relative importance of these factors may vary from one replica to the next, even for the same customer.

Functionality	DONAR API Call
create a DONAR service	`s = create ()`
add a replica instance	`i = add (s, repl, ttl)`
set split weight	`set (s, i, w`$_i$`, ε`$_i$`)`
set bandwidth cap	`set (s, i, B`$_i$`)`
match a client-replica pair	`match (s, clnt, i)`
prefer a particular replica	`preference (s, clnt, i)`
remove a replica instance	`remove (s, i)`

Figure 2: DONAR's Application Programming Interface

2.2 Application Programming Interface

Through considering a number of such policy preferences—which we review in the next section—we found that enabling customers to express two simple factors was a powerful tool. DONAR allows its customers to dictate a replica's (i) *split weight*, w_i, the desired proportion of requests that a particular replica i should receive out of the customer's total set of replicas, or (ii) *bandwidth cap*, B_i, the upper-bound on the exact number of requests that replica i should receive. In practice, different requests consume different amounts of server and bandwidth resources; we expect customers to set B_i based on the relationship between the available server/bandwidth resources and the average resources required to service a request.

These two factors enable DONAR's customers to balance load between replicas or to cap load at an individual replica. For the former, a customer specifies w_i and ε_i to indicate that it wishes replica i to receive a w_i fraction of the total request load[1], but is willing to deviate up to ε_i in order to achieve better network performance. If P_i denotes the true proportion of requests directed to i, then

$$|P_i - w_i| \leq \varepsilon_i$$

This ε_i is expressed in the same unit as the split rate; for example, if a customer wants each of its ten replicas to receive a split rate of 0.1, setting ε_* to 0.02 indicates that each replica should receive $10\% \pm 2\%$ of the request load.

Alternatively, a customer can also specify a bandwidth cap B_i per replica, to require that the exact amount of received traffic at i does not exceed B_i. If B is the total constant load across all replicas, then

$$B \cdot P_i \leq B_i$$

Note that if a set of B_i's become infeasible given a service's traffic load, DONAR reverts to w_i splits for each instance. If those are not present, excess load is spread equally amongst replicas.

Figure 2 shows DONAR's customer API. A customer creates a DONAR service by calling `create()`. A customer uses `add()` and `remove()` to add or remove a replica instance from its service's replica set. The record will persist in DONAR for the specified time-to-live period (*ttl*), unless it is explicitly removed before this period expires. A short *ttl* serves to make `add()` equivalent to a soft-state heartbeat operation, where an individual replica can execute it repeatedly to express its liveness. To communicate its preferred request distribution, the customer uses `set()` for a replica instance i. This function takes either (w_i, ε_i) or B_i, but never both

[1]In reality, the customer may select weights that do not sum to 1, particularly since each replica may assign its weight independently. DONAR simply *normalizes* the weights to sum to 1, *e.g.*, proportion $w_i / \sum_j w_j$ for replica i.

(as the combination does not make logical sense with respect to the *same* replica at the same instant in time). We do allow a customer simultaneously to use both w_i and B_i for different subsets of replicas, which we show later by example.

Some customers may want to impose more explicit constraints on specific client-replica pairs, for cost or performance reasons. For example, a customer may install a single replica inside an enterprise network for handling requests *only* from clients within that enterprise (and, similarly, the clients in the enterprise should *only* go to that server replica, no matter what the load is). A customer expresses this hard constraint using the `match()` call. Alternatively, a customer may have a strong preference for directing certain clients to a particular replica—*e.g.*, due to network peering arrangements—giving them priority over other clients. A customer expresses this policy using the `preference()` call. In contrast to `match()`, the `preference()` call is a soft constraint; in the (presumably rare) case that replica i cannot handle the total load of the high-priority clients, some of these client requests are directed to other, less-preferred replicas. These two options mimic common primitives in today's commercial CDNs.

In describing the API, we have not specified exactly who or what is initiating these API function calls—*e.g.*, a centralized service manager, individual replicas, etc.—because DONAR imposes no constraint on the source of customer policy preferences. As we demonstrate next, different use cases require updating DONAR from different sources, at different times, and based on different inputs.

2.3 Expressing Policies with DONAR's API

Customers can use this relatively simple interface to implement surprisingly complex mapping policies. It is important to note that, internally, DONAR will minimize the network cost between clients and replicas within the constraints of any customer policy. To demonstrate use of the API, we describe increasingly intricate scenarios.

Selecting the closest replica. To implement a "closest replica" policy, a customer simply sets the bandwidth caps of all replicas to infinity. That is, they impose no constraint on the traffic distribution. This update can be generated from a single source or independently from each replica.

Balancing workload between datacenters. Consider a service provider running multiple datacenters, where datacenter i has w_i servers (dedicated to this service), and jobs are primarily constrained by server resources. The service provider's total capacity at these datacenters may change over time, however, due to physical failures or maintenance, the addition of new servers or VM instances, the repurposing of resources amongst services, or temporarily bringing down servers for energy conservation. In any case, they want the proportion of traffic arriving at each datacenter to reflect current capacity. Therefore, each datacenter i locally keeps track of its number of active servers as w_i, and calls `set` (s,i,w_i,ε_i), where ε_i can be any tolerance parameter. The datacenter need only call `set()` when the number of active servers changes. DONAR then proportionally maps client requests according to these weights. This scenario is simple because (i) w_i is locally measurable within each datacenter, and (ii) w_i is easily computable, *i.e.*, decreasing w_i when servers fail and increasing when they recover.

Enforcing heuristics-based replica selection. DONAR implements a superset of the policies that are used in legacy systems to achieve heuristics-based replica selection. For example, OASIS [10] allows replicas to withdraw themselves from the server pool once they reach a given self-assessed load threshold, and then maps clients to the closest replica remaining in the pool. To realize this policy with DONAR, each replica i independently calls `set` (s,i,B_i), where B_i is the load threshold it estimates from previous history. B_i can be dynamically updated by each replica, by taking into account the prior observed performance given its traffic volume. DONAR then strictly optimizes for network cost among those replicas still under their workload threshold B_i.

	Multi-Homing	**Replicated Service**
Available Resources to Optimize	Local links with different bandwidth capacities and costs	Wide-area replicas with different bandwidth capacities and costs
Resource Allocation	Proportion of traffic to assign each link	Proportion of new clients to assign each replica

Figure 3: Multi-homed route control versus wide-area replica selection

Optimizing 95th-percentile billing costs. Network transit providers often charge based on the 95th-percentile bandwidth consumption rates, as calculated over all 5-minute or 30-minute periods in a month. To minimize such "burstable billing" costs, a distributed service could leverage an algorithm recently proposed for multi-homed route control under 95-percentile billing [21]. This algorithm, while intended for traffic engineering in multi-homed enterprise networks, could also be used to divide clients amongst service replicas; Figure 3 summarizes the relationship between these two problems. The billing optimization could be performed by whatever management system the service already runs to monitor its replicas. This system would need to track each replica's traffic load, predict future traffic patterns, run a billing cost optimization algorithm to compute the target split ratio w_i for the next time period, and output these parameters to DONAR through `set` $(s,i,w_i,0)$ calls. Of course, any mapping service capable of proportionally splitting requests (such as weighted round-robin DNS) could achieve such dynamic splits, but only by ignoring client network performance. DONAR accommodates frequently updated split ratios while still assuring that clients reach a nearby replica.

Shifting traffic to under-utilized replicas. Content providers often deploy services over a large set of geographically diverse replicas. Providers commonly want to map clients to the closest replica unless pre-determined bandwidth capacities are violated. Under such a policy, certain nodes may see too *few* requests due to an unattractive network location (*i.e.*, they are rarely the "closest node"). To remedy this problem, providers can leverage both traffic options in combination. For busy replicas, they `set` (s,i,B_i) to impose a bandwidth cap, while for unpopular replicas, they can `set` $(s,i,k,0)$, so that replica i receives at least some fixed traffic proportion k. In this way, they avoid under-utilizing instances, while offering the vast majority of clients a nearby replica.

3. REPLICA SELECTION ALGORITHMS

This section first formulates the global replica-selection problem, based on the mapping policies we considered earlier. We then propose a decentralized solution running on distributed mapping nodes. We demonstrate that each mapping node—locally optimizing the assignment of its client population and judiciously communicating with other nodes—can lead to a globally optimal assignment. We summarize the notation we use in Table 1.

3.1 Global Replica-Selection Problem

We have discussed two important policy factors that motivate our replica-selection decisions. Satisfying one of these components (*e.g.*, network performance), however, typically comes at the expense of the other (*e.g.*, accurate load distribution). As DONAR allows customers to freely express their requirements—through their

$\mathcal{N}$	Set of mapping nodes
$\mathcal{C}_n$	Set of clients for node n, $\mathcal{C}_n \subseteq \mathcal{C}$, the set of all clients
$\mathcal{I}$	Set of service replicas
R_{nci}	Proportion of traffic load that is mapped to replica i from client c by node n
α_{cn}	Proportion of node n's traffic load from client c
s_n	Proportion of total traffic load (from $\mathcal{C}$) on node n
w_i	Traffic split weight of replica i
P_i	True proportion of requests directed to replica i
B_i	Bandwidth cap on replica i
B	Total traffic load across all replicas
ε_i	Tolerance of deviation from desired traffic split

Table 1: Summary of key notations

choice of w_i, ε_i, and B_i—customers can express their willingness to trade-off performance for load distribution.

To formulate the replica-selection problem, we introduce a network model. Let $\mathcal{C}$ be the set of clients, and $\mathcal{I}$ the set of server replicas. Denote R_{ci} as the proportion of traffic from client c routed to replica i, which we solve for in our problem. The global client performance (penalty) can be calculated as

$$perf^g = \sum_{c\in\mathcal{C}} \sum_{i\in\mathcal{I}} R_{ci} \cdot cost(c,i) \tag{1}$$

Following our earlier definitions, let $P_i = \sum_{c\in\mathcal{C}} R_{ci}$ be the true proportion of requests directed to replica i. Our goal is to minimize this performance penalty, *i.e.*, to match each client with a good replica, while satisfying the customer's requirements. That goal can be expressed as the following optimization problem $\mathbf{RS^g}$:

$$\text{minimize} \quad perf^g \tag{2}$$

$$\text{subject to} \quad |P_i - w_i| \le \varepsilon_i \tag{3}$$

$$B \cdot P_i \le B_i, \forall i \tag{4}$$

B is the total amount of traffic, a constant parameter that can be calculated by summing the traffic observed at all replicas. Note that for each replica i, either constraint (3) or (4) is active, but not both.

The optimization problem can also handle the `match()` and `preference()` constraints outlined in Section 2.2. A call to `match()` imposes a *hard* constraint that client c only uses replica i, and vice versa. This is easily handled by removing the client and the replica from the optimization problem entirely, and solving the problem for the remaining clients and replicas. The `preference()` call imposes a *soft* constraint that client c has priority for mapping to replica i, assuming the replica can handle the load. This is easily handled by scaling down $cost(c,i)$ by some large constant factor so the solution maps client c to replica i whenever possible.

3.2 Distributed Mapping Service

Solving the global optimization problem directly requires a central coordinator that collects all client information, calculates the optimal mappings, and directs clients to the appropriate replicas. Instead, we solve the replica-selection problem using a *distributed* mapping service. Let $\mathcal{N}$ be the set of mapping nodes. Every mapping node $n \in \mathcal{N}$ has its own view $\mathcal{C}_n$ of the total client population, *i.e.*, $\mathcal{C}_n \subseteq \mathcal{C}$. A mapping node n receives a request from a client $c \in \mathcal{C}_n$. The node maps the client to a replica $i \in \mathcal{I}$ and returns the result to that client. In practice, each client c can represent a group of aggregated end hosts, *e.g.*, according to their postal codes. This is necessary to keep request rates per-client stable (see Section 5.2 for more discussion). Therefore, we allow freedom in directing one client to one or multiple replicas. We then expand the decision variable R_{ci} to R_{nci}, which is the fraction of traffic load that is mapped to replica i from client c by node n, *i.e.*, $\sum_i R_{nci} = 1$.

Each DONAR node monitors requests from its own client population. Let s_n be the normalized traffic load on node n (that is, n's fraction of the total load from $\mathcal{C}$). Different clients may generate different amounts of workload; let $\alpha_{cn} \in [0,1]$ denote the proportion of n's traffic that comes from client c (where $\sum_{c\in\mathcal{C}_n} \alpha_{cn} = 1$). The information of R_{nci} and α_{cn} can be measured at DONAR nodes locally, whereas collecting s_n and P_i requires either central aggregation of each node's local client load and decisions, or each node to exchange its load with its peers. The distributed node deployment also allows DONAR to check the feasibility of problem (2) easily: given local load information, calculate the total load B and determine whether (2) accepts a set of $\{w_i, B_i\}_{i\in\mathcal{I}}$ parameters.

The global replica-selection problem $\mathbf{RS^g}$, after introducing mapping nodes and the new variable R_{nci}, remains the same formulation as (2)-(4), with

$$perf^g = \sum_{n\in\mathcal{N}} s_n \sum_{c\in\mathcal{C}_n} \alpha_{cn} \sum_{i\in\mathcal{I}} R_{nci} \cdot cost(c,i) \tag{5}$$

$$P_i = \sum_{n\in\mathcal{N}} s_n \sum_{c\in\mathcal{C}_n} \alpha_{cn} \cdot R_{nci}$$

A simple approach to solving this problem is to deploy a central coordinator that collects all necessary information and calculates the decision variables for *all* mapping nodes. We seek to avoid this centralization, however, for several reasons: (i) coordination between all mapping nodes is required; (ii) the central coordinator becomes a single point-of-failure; (iii) the coordinator requires information about every client and node, leading to $O(|\mathcal{N}|\cdot|\mathcal{C}|\cdot|\mathcal{I}|)$ communication and computational complexity; and (iv) as traffic load changes, the problem needs to be recomputed and the results re-disseminated. This motivates us to develop a decentralized solution, where each DONAR node runs a distributed algorithm that is simultaneously scalable, accurate, and responsive to change.

3.3 Decentralized Selection Algorithm

We now derive a decentralized solution for the global problem $\mathbf{RS^g}$. Each DONAR node will perform a smaller-scale local optimization based on its client view. We later show how local decisions converge to the global optimum within a handful of algorithmic iterations. We leverage the theory of optimization decomposition to guide our design. Consider the global performance term $perf^g$ (5), which consists of local client performance contributed by each node:

$$perf^g = \sum_{n\in\mathcal{N}} perf^l_n$$

where

$$perf^l_n = s_n \sum_{c\in\mathcal{C}_n} \alpha_{cn} \sum_{i\in\mathcal{I}} R_{nci} \cdot cost(c,i) \tag{6}$$

Each node optimizes local performance on its client population, plus a load term imposed on the replicas. For a replica i with a split-weight constraint (the case of bandwidth-cap constraint follows similarly and is shown in the final algorithm),

$$load = \sum_{i\in\mathcal{I}} \lambda_i \left((P_i - w_i)^2 - \varepsilon_i^2 \right) \tag{7}$$

where λ_i is interpreted as the unit price of violating the constraint. We will show later how to set this value dynamically for each replica. Notice that the load associates with decision variables from all nodes. To decouple it, rewrite P_i as

$$P_i = \sum_{n\in\mathcal{N}} P_{ni} = P_{ni} + \sum_{n'\in\mathcal{N}\setminus\{n\}} P_{n'i} = P_{ni} + P_{-ni}$$

where $P_{ni} = s_n \sum_{c\in\mathcal{C}_n} \alpha_{cn} \cdot R_{nci}$ is the traffic load contributed by node n on replica i—that is, the requests from those clients directed to

Initialization

For each replica i: Set an arbitrary price $\lambda_i \geq 0$.
For each node n: Set an arbitrary decision R_{nci}.

Iteration

For each node n:
(1) Collects the latest $\{P_{n'i}\}_{i\in\mathscr{I}}$ for other n'.
(2) Collects the latest λ_i for every replica i.
(3) Solves $\mathbf{RS^l_n}$.
(4) Computes $\{P_{ni}\}_{i\in\mathscr{I}}$ and updates the info.
(5) With probability $1/|\mathscr{N}|$, for every replica i:
(6) Collects the latest $P_i = \sum_n P_{ni}$.
(7) Computes $\lambda_i \leftarrow \max\{0, \lambda_i + \theta((P_i - w_i)^2 - \varepsilon_i^2)\}$, or $\lambda_i \leftarrow \max\{0, \lambda_i + \theta((B\cdot P_i)^2 - B_i^2)\}$.
(8) Updates λ_i.
(9) Stops if $\{P_{n'i}\}_{i\in\mathscr{I}}$ from other n' do not change.

Table 2: Decentralized solution of server selection

i by DONAR node n—and P_{-ni} is the traffic load contributed by nodes other than n, independent of n's decisions.

Then the local replica selection problem for node n is formulated as the following optimization problem $\mathbf{RS^l_n}$:

$$\begin{array}{ll} \text{minimize} & perf_n^l + load_n \\ \text{variables} & R_{nci}, \forall c \in \mathscr{C}_n, i \in \mathscr{I} \end{array} \quad (8)$$

where $load_n = load, \forall n$. To solve this local problem, a mapping node needs to know (i) local information about s_n and α_{cn}, (ii) prices λ_i for all replicas, and (iii) the aggregated P_{-ni} information from other nodes. Equation (8) is a quadratic programming problem, which can be solved efficiently by standard optimization solvers (we evaluate computation time in Section 5.1).

We formally present the decentralized algorithm in Table 2. At the initialization stage, each node picks an arbitrary mapping decision, *e.g.*, one that only optimizes local performance. Each replica sets an arbitrary price, say $\lambda_i = 0$. The core components of the algorithm are the local updates by each mapping node, and the periodic updates of replica prices. Mapping decisions are made at each node n by solving $\mathbf{RS^l_n}$ based on the latest information. Replica prices (λ_i) are updated based on the inferred traffic load to each i. Intuitively, the price increases if i's split weight or bandwidth requirement is violated, and decreases otherwise. This can be achieved via additive updates (shown by θ). We both collect and update the P_{ni} and λ_i information through a data store service, as discussed later.

While the centralized solution requires $O(|\mathscr{N}|\cdot|\mathscr{C}|\cdot|\mathscr{I}|)$ communication and computation at the coordinator, the distributed solution has much less overhead. Each node needs to share its mapping decisions of size $|\mathscr{I}|$ with all others, and each replica's price λ_i needs to be known by each node. This implies $|\mathscr{N}|$ messages, each of size $O((|\mathscr{N}|-1)\cdot|\mathscr{I}|+|\mathscr{I}|) = O(|\mathscr{N}|\cdot|\mathscr{I}|)$. Each node's computational complexity is of size $O(|\mathscr{C}_n|\cdot|\mathscr{I}|)$.

The correctness of the distributed algorithm relies on an appropriate ordering of local updates from each node, *i.e.*, in a round-robin fashion as shown in Appendix A, and a less frequent replica price update. In practice, however, we allow nodes and replicas to update *uncoordinatedly* and *independently*. We find that the algorithm's convergence is not sensitive to this lack of coordination, which we demonstrate in our evaluation. In fact, the decentralized solution works well even at the scale of thousands of mapping nodes. For a given replica-selection problem, the decentralized solution usually converges within a handful of iterations, and the equilibrium point is also the optimal solution to the global problem. Appendix A provides the derivation of this decentralized solution.

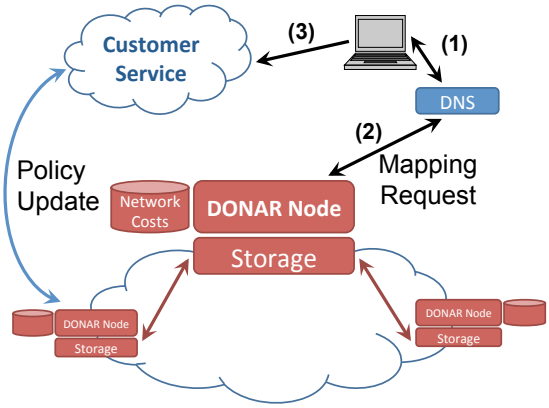

Figure 4: Interactions on a DONAR node

4. DONAR'S SYSTEM DESIGN

This section describes the design of DONAR, which provides distributed replica selection for large numbers of customers, each of whom have their own set of service replicas and different high-level preferences over replica selection criteria. DONAR implements the policy interface and distributed optimization mechanism we defined in the last two sections. Each DONAR node must also reliably handle client requests and customer updates. DONAR nodes should be geographically dispersed themselves, for greater reliability and better performance. Our current deployment, for example, consists of globally dispersed machines on both the PlanetLab [22] and VINI [23] network platforms.

Figure 4 depicts a single DONAR node and its interactions with various system components. This section is organized around these components. Section 4.1 discusses how DONAR nodes combine customer policies, mapping information shared by other nodes, and locally-available cost information, in order to optimally map clients to customers' service replicas. Section 4.2 describes DONAR's front-end mechanisms for performing replica selection (*e.g.*, DNS, HTTP redirection, HTTP proxying, etc.), and Section 4.3 details its update protocol for registering and updating customer services, replicas, and policies. Finally, Section 4.4 describes DONAR's back-end distributed data store (for reliably disseminating data) and use of IP Anycast (for reliably routing client requests).

4.1 Efficient Distributed Optimization

Given policy preferences for each customer, DONAR nodes must translate high-level mapping goals into a specific set of rules for each node. DONAR's policy engine realizes a variant of the algorithm described in Table 2. Particularly, all nodes act asynchronously in the system, so no round-based coordination is required. We demonstrate in Section 5 that this variant converges in practice.

Request rate and cost estimation. Our model assumes that each node has an estimate of the request rate per client. As suggested in Section 3, each "client" represents a group of similarly located end-hosts (we also refer to this entity as a "client region"). When a DONAR node comes online and begins receiving client requests, it tracks the request volume per unit time per client region. While our optimization algorithm models a static problem — and therefore constant request rates — true request rates vary over time. To address this limitation, DONAR nodes use an exponentially-weighted moving average of previous time intervals (with $\alpha = .8$ and 10 minute intervals in our current deployment). Still, rapid changes

in a particular client region might lead to suboptimal mapping decisions. Using trace data from a popular CDN, however, we show in Section 5.2 that relative request rates per region do not significantly vary between time intervals.

Our model also assumes a known *cost(c,i)* function, which quantifies the cost of pairing client c with instance i. In our current deployment, each DONAR node has a commercial-grade IP geolocation database [12] that provides this data and is updated weekly. We use a *cost(c, i)* function that is the normalized Euclidean distance between the IP prefixes of the client c and instance i (like that described in Section 3).

Performing local optimization. DONAR nodes arrive at globally optimal routing behavior by periodically re-running the local optimization problem. In our current implementation, nodes each run the same local procedure at regular intervals (about every 2 minutes, using some randomized delay to desynchronize computations).

As discussed in Section 3.3, the inputs to each local optimization are the aggregate traffic information sent by all remote DONAR nodes $\{B \cdot P_{-ni}\}_{i \in \mathscr{I}}$; the customer policy parameters, $\{w_i, B_i\}_{i \in \mathscr{I}}$; and the proportions of local traffic coming from each client region to that node, $\{\alpha_{cn}\}_{c \in \mathscr{C}_n}$. The first two inputs are available through DONAR's distributed data store, while nodes locally compute their clients' proportions. Given these inputs, the node must decide how to best map its clients. This reduces to the local optimization problem $\mathbf{RS^l_n}$, which minimizes a node's total client latency given the constraints of the customer's policy specification and the mapping decisions of other nodes (treating their most recent update as a static assignment). The outcome of this optimization is a new set of local rules $\{R_{nci}\}_{c \in \mathscr{C}_n i \in \mathscr{I}}$, which dictates the node's mapping behavior. Given these new mapping rules, the node now expects to route different amounts of traffic to each replica. It then computes these new expected traffic rates per instance, using the current mapping policy and its historical client request rates $\{\alpha_{cn}\}_{c \in \mathscr{C}_n}$. It then updates its existing per-replica totals $\{B \cdot P_i\}_{i \in \mathscr{I}}$ in the distributed data store, so that implications of its new local policy propagate to other nodes.

If the new solution violates customer constraints — bandwidth caps are overshot or split's exceed the allowed tolerance — the node will update the constraint multipliers $\{\lambda_i\}_{i \in \mathscr{I}}$ with probability of $1/|\mathscr{N}|$. Thus, in the case of overload, the multipliers will be updated, on average, once per cycle of local updates.

4.2 Providing Flexible Mapping Mechanisms

A variety of protocol-level mechanisms are employed for wide-area replica selection today. They include (i) dynamically generated DNS responses with short TTLs, according to a given policy, (ii) using HTTP Redirection from a centralized source and/or between replicas, and (iii) using persistent HTTP proxies to tunnel requests.

To offer customers maximum flexibility, DONAR offers all three of these mechanisms. To use DONAR via DNS, a domain's owner will register each of its replicas as a single A record with DONAR, and then point the NS records for its domain (*e.g.*, `example.com`) to `ns.donardns.org`. DONAR nameservers will then respond to requests for `example.com` with an appropriate replica given the domain's selection criteria.

To use DONAR for HTTP redirection or proxying, a customer adds HTTP records to DONAR—a record type in DONAR update messages—such as mapping `example.com` to `us-east.example.com`, `us-west.example.com`, etc. DONAR resolves these names and appropriately identifies their IP address for use during its optimization calculations.[2] This customer then hands off DNS authority to DONAR as before. When DONAR receives a DNS query for the domain, it returns the IP address of the client's nearest DONAR node. Upon receiving the corresponding HTTP request, a DONAR HTTP server uses requests' `Host:` header fields to determine for which customer domains their requests correspond. It queries its local DONAR policy engine for the appropriate replica, and then redirects or proxies the request to that replica.

The DONAR software architecture is designed to support the easy addition of new protocols. DONAR's DNS nameserver and HTTP server run as separate processes, communicating with the local DONAR policy engine via a standardized socket protocol. Section 4.5 describes additional implementation details.

[2]In this HTTP example, customers need to ensure that each name resolves either to a single IP address or a set of collocated replicas.

4.3 Secure Registration and Dynamic Updates

Since DONAR aims to accommodate many simultaneous customers, it is essential that all customer-facing operations be completely automated and not require human intervention. Additionally, since DONAR is a public service, it must authenticate client requests and prevent replay attacks. To meet these goals we have developed a protocol which provides secure, automatic account creation and facilitates frequent policy updates.

Account creation. In DONAR, a customer account is uniquely identified by a private/public key pair. DONAR reliably stores its customers' public keys, which are used to cryptographically verify signatures on account updates (described next). Creating accounts in DONAR is completely automated, *i.e.*, no central authority is required to approve account creation. To create a DONAR account, a customer generates a public/private key-pair and simply begins adding new records to DONAR, signed with the private key. If a DONAR node sees an unregistered public key in update messages, it generates the SHA-1 hash of the key, `hash`, and allocates the domain `<hash>.donardns.net` to the customer.

Customers have the option of validating a domain name that they own (*e.g.*, `example.com`). To do so, a customer creates a temporary CNAME DNS record that maps `validate-<hash>.example.com` to `donardns.net`. Since only someone authoritative for the example.com namespace will be able to add this record, its presence alone is a sufficient proof of ownership. The customer then sends a validation request for `example.com` to a DONAR node. DONAR looks for the validation CNAME record in DNS and, if found, will replace `<hash>.donardns.net` with `example.com` for all records tied to that account.

DONAR Update Protocol (DUP). Customers interact with DONAR nodes through the DONAR Update Protocol (DUP). Operating over UDP, DUP allows customers to add and remove new service replicas, express the policy parameters for these replicas as described in Section 3, as well as implicitly create and explicitly verify accounts, per above. DUP is similar in spirit to the DNS UPDATE protocol [24] with some important additional features. These include mandatory RSA signatures, nonces for replay protection, DONAR-specific meta-data (such as *split weight* or *bandwidth cap*), and record types outside of DNS (for HTTP mapping). DUP is record-based (like DNS), allowing forward compatibility as we add new features such as additional policy options.

4.4 Reliability through Decentralization

DONAR provides high availability by gracefully tolerating the failure of individual DONAR nodes. To accomplish this, DONAR incorporates reliable distributed data storage (for customer records) and ensures that clients will be routed away from failed nodes.

Distributed Data Storage. DONAR provides distributed storage of customer record data. A customer update (through DUP) should

be able to be received and handled by *any* DONAR node, and the update should then be promptly visible throughout the DONAR network. There should be no central point of failure in the storage system, and the system should scale-out with the inclusion of new DONAR nodes without any special configuration.

To provide this functionality, DONAR uses the CRAQ storage system [25] to replicate record data and account information across all participating nodes. CRAQ automatically re-replicates data under membership changes and can provide either strong or eventual consistency of data. Its performance is optimized for read-heavy workloads, which we expect in systems like DONAR where the number of client requests likely will be orders of magnitude greater than the number of customer updates. DONAR piggybacks on CRAQ's group-membership functionality, built on Zookeeper [26], in order to alert DONAR nodes when a node fails. While such group notifications are not required for DONAR's availability, this feature allows DONAR to quickly recompute and reconverge to optimal mapping behavior following node failures.

Route control with IP anycast. While DONAR's storage system ensures that valuable data is retained in the face of node failures, it does not address the issue of routing client requests away from failed nodes. When DONAR is used for DNS, it can partially rely on its clients' resolvers performing automatic failover between authoritative nameservers. This failover significantly increases resolution latency, however. Furthermore, DONAR node failure presents an additional problem when used with HTTP-based selection. Most web browsers do not failover between multiple A records (in this case, DONAR HTTP servers), and browsers—and now browser plugins like Java and Flash as well—purposely "pin" DNS names to specific IP addresses to prevent "DNS rebinding" attacks [27].[3] These cached host-to-address bindings often persist for several minutes. To address both cases, DONAR is designed to work over IP anycast, not only for answering DNS queries but also for processing updates.

In our current deployment, a subset of DONAR nodes run on VINI [23], a private instance of PlanetLab which allows tighter control over the network stack. These nodes run an instance of Quagga that peers with TransitPortal instances at each site [28], and thus each site announces DONAR's /24 prefix through BGP. If a node loses connectivity, the BGP session will drop and the Transit Portal will withdraw the wide-area route. To handle application-level failures, a watchdog process on each node monitors the DONAR processes and withdraws the BGP route if a critical service fails.

4.5 Implementation

The software running on each DONAR node consists of several modular components which are detailed in Figure 5. They constitute a total of approximately 10,000 lines of code (C++ and Java), as well as another 2,000 lines of shell scripts for system management.

DONAR has been running continuously on Measurement Lab (M-Lab) since October 2009. All services deployed on M-Lab have a unique domain name:

`⟨service⟩.⟨account⟩.donar.measurement-lab.org`

that provides a closest-node policy by default, selecting from among the set of M-Lab servers. Two of the most popular M-Lab services—the Network Diagnostic Tool (NDT) [29], which is used for the Federal Communication Commission's Consumer Broadband Test, and NPAD [30]—are more closely integrated with DONAR. NDT and NPAD run the DUP protocol, providing DONAR nodes with status updates every 5 minutes.

[3]A browser may pin DNS–IP mappings even after observing destination failures; otherwise, an attacker may forge ICMP "host unreachable" messages to cause it to unpin a specific mapping.

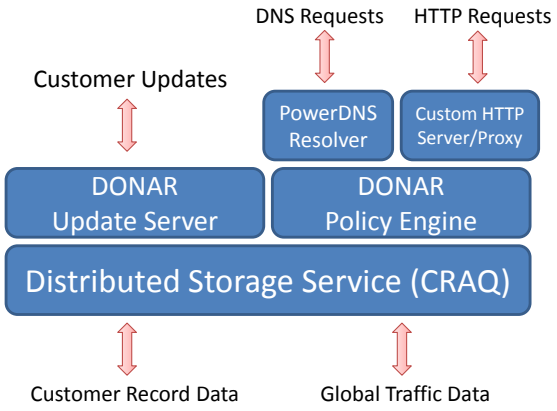

Figure 5: Software architecture of a DONAR node

Since December of 2009, DONAR has also handled 15% CoralCDN's DNS traffic [14], around 1.25 million requests per day. This service uses an equal-split policy amongst its replicas.

DONAR's current implementation supports three front-end mapping mechanisms, implemented as separate processes for extensibility, which communicate via the main DONAR policy engine over a UNIX domain socket and a record-based ASCII protocol. The DNS front-end is built on the open-source PowerDNS resolver, which supports customizable storage backends. DONAR provides a custom HTTP server for HTTP redirection, built on top of the libmicrohttpd embedded HTTP library. DONAR also supports basic HTTP tunneling *i.e.*, acting as a persistent proxy between clients and replicas, via a custom built HTTP proxy. However, due to the bandwidth constraints of our deployment platform, it is currently disabled.

At the storage layer, DONAR nodes use CRAQ to disseminate customer records (public key, domain, and replica information), as well as traffic request rates to each service replica (from each DONAR node). CRAQ's key-value interface offers basic set/get functionality; DONAR's primitive data types are stored with an XDR encoding in CRAQ.

The DONAR policy engine is written in C++, built using the Tame extensions [31] to the SFS asynchronous I/O libraries. In order to assign client regions to replica instances, the engine solves a quadratic program of size $|\mathscr{C}_n| \cdot |\mathscr{I}|$, using a quadratic solver from the MOSEK [32] optimization library.

The DONAR update server, written in Java, processes DUP requests from customers, including account validation, policy updates, and changes in the set of active replicas. It uses CRAQ to disseminate record data between nodes. While customers can build their own applications that directly speak DUP, we also provide a publicly-available Java client that performs these basic operations.

5. EVALUATION

Our evaluation of DONAR is in three parts. Section 5.1 simulates a large-scale deployment given trace request data and simulated mapping nodes. Section 5.2 uses the same dataset to verify that client request volumes are reasonably stable from one time period to the next. Finally, Section 5.3 evaluates our prototype performing real-world replica selection for a popular CDN.

5.1 Trace-Based Simulation

We use trace data to demonstrate the performance and stability of our decentralized replica-selection algorithm. We analyzed DNS

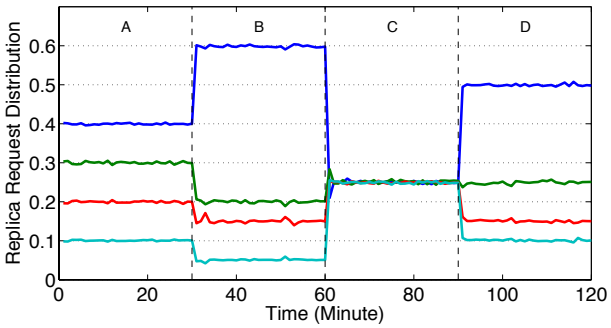

Figure 6: DONAR adapts to split weight changes

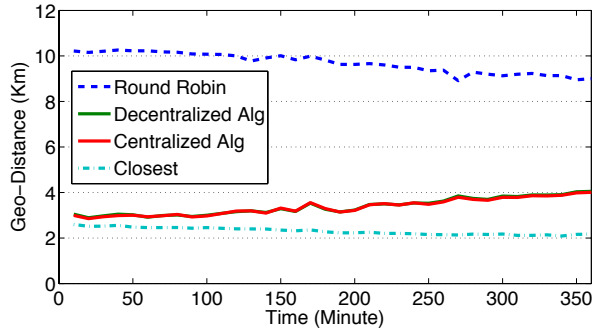

Figure 7: Network performance implications of replica-selection policies

log files from CoralCDN. Our dataset consists of 9,918,780 requests over a randomly-selected 24-hour period (July 28, 2009). On that day, CoralCDN's infrastructure consisted of 76 DNS servers which dispatched clients to any of 308 HTTP proxies distributed world-wide. Locality information was obtained through Quova's commercial geolocation database [12].

In the trace-based simulation, we aggregate clients by geographic location. This results in 371 distinct client regions in our trace. We choose 10 hypothetical mapping nodes to represent a globally-distributed set of authoritative nameservers. Each request is assigned to the nearest nameserver, partitioning the client space. Four replica instances are selected from locations in the US-East, US-West, Europe, and Asia.

We feed each mapping node with client request rates and distributions, which are inferred from the trace for every 10-minute interval. In the evaluation, we allow DONAR nodes to communicate *asynchronously*, *i.e.*, they do not talk in a circular fashion as in Table 2. Instead, we overlap updates such that there is 40% probability that at least half of nodes update simultaneously. All nodes perform an update once each minute. We use the quadratic programming solver in MOSEK [32], and each local optimization takes about 50ms on a 2.0GHz dual core machine.

Load split weight. Customers can submit different sets of load split weights over time, and we show how DONAR dynamically adapts to such changes. Figure 6 shows the replica request distribution over a 2-hour trace. We vary the desired split weight four times, at 0, 30, 60 and 90 minutes. Phase A shows replica loads quickly converging from a random initial point to a split weight of $40/30/20/10$. Small perturbations occur at the beginning of every 10 minutes, since client request rates and distributions change. Replica loads quickly converge to the original level as DONAR re-optimizes based on the current load. In Phase B, we adjust the split weight to $60/20/15/5$, and replica loads shift to the new level usually within 1 or 2 minutes. Note that the split weight and traffic load can change simultaneously, and DONAR is very responsive to these changes. In Phase C, We implement an equal-split policy and Phase D re-balances the load to an uneven distribution. In this experiment we chose $\varepsilon_i = 0.01$, so there is very little deviation from the exact split weight. This example demonstrates the nice responsiveness and convergence property of the decentralized algorithm, even when the local optimizations run asynchronously.

	% of clients to closest replica	mean	variance
$\varepsilon_i = 0$	54.88%	4.0233	0.0888
$\varepsilon_i = 1\%$	69.56%	3.3662	0.1261
$\varepsilon_i = 5\%$	77.13%	3.0061	0.1103
$\varepsilon_i = 10\%$	86.34%	2.8690	0.4960
closest replica	100%	2.3027	0.0226

Figure 8: Sensitivity analysis of using tolerance parameter ε_i

Network performance. We next investigate the network performance under an equal-split policy among replicas, *i.e.*, all four replicas expect 25% of load and tolerate $\varepsilon_i = 1\%$ deviation. We use a 6-hour trace from the above dataset, starting at 9pm EST. We compare DONAR's decentralized algorithm to three other replica-selection algorithms. *Round Robin* maps incoming requests to the four replicas in a round-robin fashion, achieving equal load distribution. *Centralized Alg* uses a central coordinator to calculate mapping decision for all nodes and all clients, and thus does not require inter-node communication. *Closest* always maps a client to the closest replica and achieves the best network performance. The performance of these algorithms is shown in Figure 7. The best (minimum) distance, realized by Closest, is quite stable over time. Round Robin achieves the worst network performance, about 300%–400% more than the minimum, since 75% of requests go to sub-optimal replicas. DONAR's decentralized algorithm can achieve much better performance, realizing 10%–100% above the minimum distance in exchange for better load distribution. Note that the decentralized solution is very close to that of a central coordinator.

It is also interesting to note that DONAR's network cost is increasing. This can be explained by diurnal patterns in different areas: the United States was approaching midnight while Asia reached its traffic peak at noon. Requiring 25% load at each of the two US servers understandably hurts the network performance.

Sensitivity to tolerance parameter. When submitting split weights, a customer can use ε_i to strike a balance between strict load distribution and improved network performance. Although an accurate choice of ε_i depends on the underlying traffic load, we use our trace to shed some light on its usage. In Figure 8, we employ an equal-split policy, and try $\varepsilon_i = 0, 1\%, 5\%$ and 10%. We show the percentage of clients that are mapped to the closest replica, and the mean performance and variance, over the entire 6-hour trace. We also compare them to the closet-replica policy. Surprisingly, tolerating a 1% deviation from a strict equal split allows 15% more clients to map to the closest replica. 5% tolerance can further improve 7% of nodes, and an additional 9% improvement is possible for a 10% tolerance. This demonstrates that ε_i provides a very tangible mechanism for trading off network performance and traffic distribution.

5.2 Predictability of Client Request Rate

So far we have shown rapid convergence given a temporarily fixed set of request rates per client. In reality, client request volume will vary, and DONAR's *predicted* request volume for a given client may not accurately forecast client traffic. For DONAR to

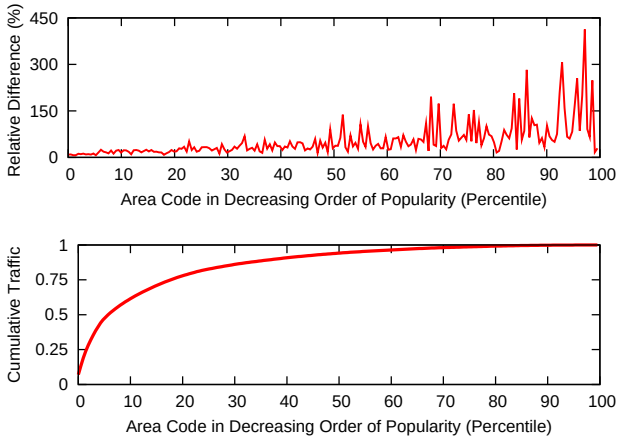

Figure 9: Stability of area code request rates

work well, client traffic rates must be sufficiently predictable under a granularity that remains useful to our customers. Our current deployment uses a fixed interval size of 10 minutes. We now show via analysis of the same trace data, that request rates are sufficiently predictable on this timescale.

Figure 9 (top) plots the relative difference between our estimated rate and the true rate for each client group, *i.e.*, a value of zero indicates a perfect prediction of request volume. Each data point is the average difference over a 2-hour interval for one client. Figure 9 (bottom) is a CDF of all traffic from these same clients, which shows that the vast majority of incoming traffic belongs to groups whose traffic is very stable. The high variation in request rate of the last 50% of groups accounts for only 6% of total traffic.

Coarser-grained client aggregation (*i.e.*, larger client groups) will lead to better request stability, but at the cost of locality precision. In practice, services prefer a fine granularity in terms of rate intervals and client location. Commercial CDN's and other replicated services, which see orders-of-magnitude more traffic than CoralCDN, would be able to achieve much finer granularity while keeping rates stable, such as tracking requests per minute, per IP prefix.

5.3 Prototype Evaluation

We now evaluate our DONAR prototype when used to perform replica selection for CoralCDN. CoralCDN disseminates content by answering HTTP requests on each of its distributed replicas. Since clients access CoralCDN via the `nyud.net` domain suffix, they require a DNS mechanism to perform replica selection. For our experiments, we create a DONAR account for the `nyud.net` suffix, and add ten CoralCDN proxies as active replicas. We then point a subset of the CoralCDN NS records to DONAR's mapping nodes in order to direct DNS queries.

Closest Replica. We first implement a "closest replica" policy by imposing no constraint on the distribution of client requests. We then track the client arrival rate at each replica, calculated in 10-minute intervals over the course of three days. As Figure 10 demonstrates, the volume of traffic arriving at each replica varies highly according to replica location. The busiest replica in each interval typically sees ten times more traffic than the least busy. For several periods a single server handles more than 40% of requests. The diurnal traffic fluctuations, which are evident in the graph, also increase the variability of traffic on each replica.

Each CoralCDN replica has roughly the same capacity, and each replica's performance diminishes with increased client load, so a

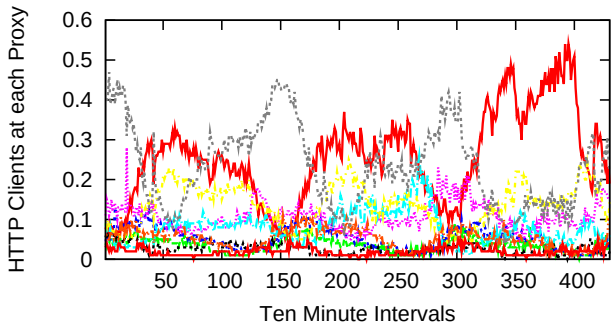

Figure 10: Server request loads under "closest replica" policy

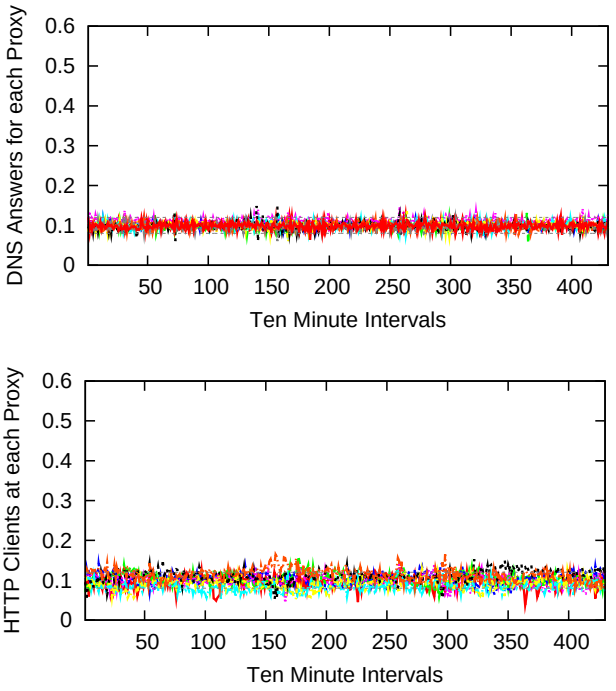

Figure 11: Proportional traffic distribution observed by DONAR (Top) and CoralCDN (Bottom), when an equal-split policy is enacted by DONAR. Horizontal gray lines represent the ε tolerance $\pm 2\%$ around each split rate.

preferable outcome is one in which loads are relatively uniform. Despite a globally distributed set of replicas serving distributed clients, a naïve replica selection strategy results in highly disproportionate server loads and fails to meet this goal. Furthermore, due to diurnal patterns, there is no way to statically provision our servers in order to equalize load under this type of selection policy. Instead, we require a dynamic mapping layer to balance the goals of client proximity and load distribution, as we show next.

Controlling request distribution. We next leverage DONAR's API to dictate a specific distribution of client traffic amongst replicas. In this evaluation we partition the ten Coral servers to receive equal amounts of traffic (10% each) each with an allowed deviation of 2%. We then measure both the mapping behavior of each DONAR node and the client arrival rate at each CoralCDN replica. Figure 11 demonstrates the proportion of requests mapped to each replica as recorded by DONAR nodes. Replica request volumes fluctuate within the allowed range as DONAR adjusts to changing client request patterns. The few fluctuations which extend past the

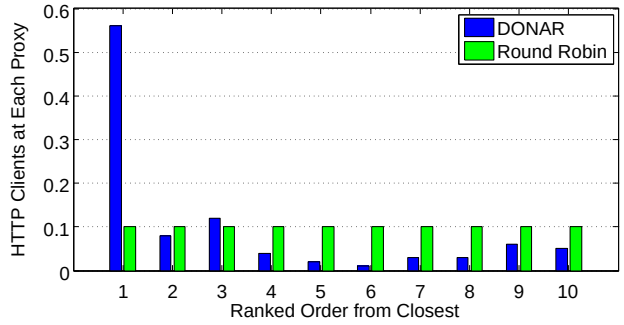

Figure 12: Client performance during equal split

"allowed" tolerance are due to inaccuracies in request load prediction and intermediate solutions which (as explained in Section 3) may temporarily violate constraints. Figure 11 (Bottom) depicts the arrival rate of clients at each CoralCDN node. The discrepancy in these graphs is an artifact of the choice to use DNS-based mapping, a method which offers complete transparency at the cost of some control over mapping behavior (as discussed in Section 6). Nonetheless, request distribution remains within 5% of desired during most time periods. Customers using HTTP-based mechanisms would see client arrival exactly equal to that observed by DONAR.

Measuring client performance. We next evaluate the performance of CoralCDN clients in the face of specific the traffic distribution requests imposed in the prior experiment. Surprisingly, DONAR is able to sufficiently balance client requests, while pairing clients with nearby servers with very high probability. This distinguishes DONAR from simple weighted load balancing schemes, which forgo a consideration of network performance in favor of achieving a specific split. While DONAR nodes define fractional rules for each client, in practice nearly every client is "pinned" to a particular replica at optimum. We can thus show the proportion of clients paired with their n^{th} preferred replica. Figure 12 plots this data, contrasting DONAR with a round-robin mapping policy. With DONAR, more than 50% of clients are mapped to the nearest node and 75% are mapped within the top three, a significant improvement over the traditional round-robin approach.

6. RELATED WORK

Network distance estimation. A significant body of research has examined techniques for network latency and distance estimation, useful for determining `cost(c,i)` in DONAR. Some work focused on reducing the overhead for measuring the IP address space [33, 19, 34, 35]. Alternatively, virtual coordinate systems [17, 18] estimated latency based on synthetic coordinates. More recent work considered throughput, routing problems, and abnormalities as well [20, 16]. Another direction is geographic mapping techniques, including `whois` data [36, 37], or extracting information from location-based naming conventions for routers [38, 39].

Configurable Mapping Policies. Existing services provide limited customization of mapping behavior. OASIS [10] offers a choice of two heuristics which jointly consider server load and client-server proximity. ClosestNode [40] supports locality policies based on programmatical on-demand network probing [15]. Amazon's EC2, a commercial service, allows basic load balancing between virtual machine instances. For those who are willing to set up their own distributed DNS infrastructure, packages like MyXDNS [41] facilitate extensive customization of mapping behavior, though only by leaving all inputs and decisions up to the user.

Efficacy of Request Redirection. Several studies have evaluated the effectiveness of DNS-based replica selection, including the accuracy of using DNS resolvers to approximate client location [42] or the impact of caching on DNS responsiveness [43]. Despite these drawbacks, DNS remains the preferred mechanism for sever selection by many industry leaders, such as Akamai [2]. While HTTP tunneling and request redirection offer greater accuracy and finer-grain control for services operating on HTTP, they introduce additional latency and overhead.

7. CONCLUSIONS

DONAR enables online services to offer better performance to their clients at a lower cost, by directing client requests to appropriate server replicas. DONAR's expressive interface supports a wide range of replica-selection policies, and its decentralized design is scalable, accurate, and responsive. Through a live deployment with CoralCDN, we demonstrate that our distributed algorithm is accurate and efficient, requiring little coordination among the mapping nodes to adapt to changing client demands. In our ongoing work, we plan to expand our CoralCDN and M-Lab deployments and start making DONAR available to other services. In doing so, we hope to identify more ways to meet the needs of networked services.

Acknowledgments. The authors are grateful to Andrew Schran for the design and prototype of the Namecast service [44] that was a precursor to DONAR, and to Quova, Inc. for access to their IP2Geo database. Thanks also to Eric Keller, Wyatt Lloyd, Kay Ousterhout, Sid Sen, the anonymous SIGCOMM reviewers, and our shepherd, Jinyang Li, for their comments on earlier versions of the paper.

References

[1] Amazon Web Services, "http://aws.amazon.com/," 2010.
[2] Akamai Technologies. http://www.akamai.com/, 2010.
[3] AmazonAWS, "Elastic load balancing." http://aws.amazon.com/ elasticloadbalancing/, 2010.
[4] DynDNS. http://www.dyndns.com/, 2010.
[5] UltraDNS. http://www.ultradns.com/, 2010.
[6] B. Maggs, "Personal communication," 2009.
[7] M. Colajanni, P. S. Yu, and D. M. Dias, "Scheduling algorithms for distributed web servers," in *ICDCS*, May 1997.
[8] M. Conti, C. Nazionale, E. Gregori, and F. Panzieri, "Load distribution among replicated Web servers: A QoS-based approach," in *Workshop Internet Server Perf.*, May 1999.
[9] V. Cardellini, M. Colajanni, and P. S. Yu, "Geographic load balancing for scalable distributed web systems," in *MASCOTS*, Aug. 2000.
[10] M. J. Freedman, K. Lakshminarayanan, and D. Mazières, "OASIS: Anycast for any service," in *NSDI*, May 2006.
[11] M. Pathan, C. Vecchiola, and R. Buyya, "Load and proximity aware request-redirection for dynamic load distribution in peering CDNs," in *OTM*, Nov. 2008.
[12] Quova. http://www.quova.com/, 2010.
[13] MeasurementLab. http://www.measurementlab.net/, 2010.
[14] M. J. Freedman, E. Freudenthal, and D. Mazières, "Democratizing content publication with Coral," in *NSDI*, Mar. 2004.
[15] B. Wong, A. Slivkins, and E. G. Sirer, "Meridian: A lightweight network location service without virtual coordinates," in *SIGCOMM*, Aug. 2005.
[16] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao, "Moving beyond end-to-end path information to optimize CDN performance," in *SIGCOMM*, Aug. 2009.
[17] E. Ng and H. Zhang, "Predicting Internet network distance with coordinates-based approaches," in *INFOCOM*, June 2002.

[18] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *SIGCOMM*, Aug. 2004.
[19] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "IDMaps: A global Internet host distance estimation service," *Trans. Networking*, Oct. 2001.
[20] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iPlane: An information plane for distributed services," in *OSDI*, Nov. 2006.
[21] D. K. Goldenberg, L. Qiu, H. Xie, Y. R. Yang, and Y. Zhang, "Optimizing cost and performance for multihoming," in *SIGCOMM*, Aug. 2004.
[22] "PlanetLab." http://www.planet-lab.org/, 2008.
[23] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In VINI veritas: Realistic and controlled network experimentation," in *SIGCOMM*, Aug. 2006.
[24] S. Thomson, Y. Rekhter, and J. Bound, "Dynamic updates in the domain name system (DNS UPDATE)," 1997. RFC 2136.
[25] J. Terrace and M. J. Freedman, "Object storage on CRAQ: High-throughput chain replication for read-mostly workloads," in *USENIX Annual*, June 2009.
[26] Zookeeper. http://hadoop.apache.org/zookeeper/, 2010.
[27] D. Dean, E. W. Felten, and D. S. Wallach, "Java security: From HotJava to Netscape and beyond," in *Symp. Security and Privacy*, May 1996.
[28] V. Valancius, N. Feamster, J. Rexford, and A. Nakao, "Wide-area route control for distributed services," in *USENIX Annual*, June 2010.
[29] Internet2, "Network diagnostic tool (ndt)." http://www.internet2.edu/performance/ndt/, 2010.
[30] M. Mathis, J. Heffner, and R. Reddy, "Network path and application diagnosis (npad)." http://www.psc.edu/networking/projects/pathdiag/, 2010.
[31] M. Krohn, E. Kohler, and F. M. Kaashoek, "Events can make sense," in *USENIX Annual*, Aug. 2007.
[32] MOSEK, "http://www.mosek.com/," 2010.
[33] J. Guyton and M. Schwartz, "Locating nearby copies of replicated Internet servers," in *SIGCOMM*, Aug. 1995.
[34] W. Theilmann and K. Rothermel, "Dynamic distance maps of the Internet," in *IEEE INFOCOM*, Mar. 2001.
[35] Y. Chen, K. H. Lim, R. H. Katz, and C. Overton, "On the stability of network distance estimation," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 2, pp. 21–30, 2002.
[36] 2005. http://cello.cs.uiuc.edu/cgi-bin/slamm/ip2ll/.
[37] D. Moore, R. Periakaruppan, and J. Donohoe, "Where in the world is netgeo.caida.org?," in *INET*, June 2000.
[38] V. N. Padmanabhan and L. Subramanian, "An investigation of geographic mapping techniques for Internet hosts," in *SIGCOMM*, Aug. 2001.
[39] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan, "Geographic locality of IP prefixes," in *IMC*, Oct. 2005.
[40] B. Wong and E. G. Sirer, "ClosestNode.com: An open access, scalable, shared geocast service for distributed systems," *SIGOPS OSR*, vol. 40, no. 1, 2006.
[41] H. A. Alzoubi, M. Rabinovich, and O. Spatscheck, "MyXDNS: A resquest routing DNS server with decoupled server selection," in *WWW*, May 2007.
[42] Z. M. Mao, C. D. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang, "A precise and efficient evaluation of the proximity between Web clients and their local DNS servers," in *USENIX Annual*, June 2002.
[43] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan, "On the responsiveness of DNS-based network control," in *IMC*, Oct. 2004.
[44] A. Schran, J. Rexford, and M. J. Freedman, "Namecast: A reliable, flexible, scalable DNS hosting system," Tech. Rep. TR-850-09, Princeton University, Apr. 2009.
[45] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.
[46] D. P. Bertsekas, *Nonlinear Programming*. Athena Scientific, 1999.

APPENDIX

A. DECENTRALIZED SOLUTION PROOF

Theorem 1 *The distributed algorithm shown in Table 2, converges to the optimal solution of* $\mathbf{RS^g}$*, given that (i) each node n iteratively solves* $\mathbf{RS^l_n}$ *in a circular fashion,* i.e.*,* $n = 1,2,\ldots|\mathcal{N}|,1,\ldots$*, and (ii) each replica price* λ_i *is updated in a larger timescale,* i.e.*, after all nodes' decisions converge given a set of* $\{\lambda_i\}_{i\in\mathcal{I}}$*.*

Proof: It suffices to show that the distributed algorithm is an execution of the dual algorithm that solves $\mathbf{RS^g}$. We only show the case of split weight constraint (3), and the case of bandwidth cap constraint (4) would follow similarly.

First, following the Lagrangian method for solving an optimization problem, we derive the Lagrangian of the global problem $\mathbf{RS^g}$. Constraint (3) is equivalent to

$$(P_i - w_i)^2 \le \varepsilon_i^2$$

The Lagrangian of $\mathbf{RS^g}$ is written as:

$$\begin{aligned} L(R,\lambda) &= perf^g + \sum_{i\in\mathcal{I}} \lambda_i\big((P_i-w_i)^2 - \varepsilon_i^2\big) \\ &= \sum_{n\in\mathcal{N}} perf_n^l + load \end{aligned} \tag{9}$$

where $\lambda_i \ge 0$ is the Lagrange multiplier (replica price) associated with the split weight constraint on replica i, and $R = \{R_{nci}\}_{n\in\mathcal{N},c\in\mathcal{C}_n,i\in\mathcal{I}}$ is the primal variable. The dual algorithm requires to minimize the Lagrangian (9) for a given set of $\{\lambda_i\}_{i\in\mathcal{I}}$.

$$\begin{array}{ll} \text{minimize} & L(R,\lambda) \\ \text{variable} & R \end{array}$$

A distributed algorithm implied by condition (i) solves (9), because each node n iteratively solving $\mathbf{RS^l_n}$ in a circular fashion, simply implements the nonlinear Guass-Seidel algorithm (per [45, Ch. 3, Prop. 3.9], [46, Prop. 2.7.1]):

$$R_n^{(t+1)} = \operatorname{argmin} \mathbf{RS^g}(\ldots, R_{n-1}^{(t+1)}, R_n, R_{n+1}^{(t)}, \ldots)$$

where $R_n = \{R_{nci}\}_{c\in\mathcal{C}_n,i\in\mathcal{I}}$ denotes node n's decision variable. Given a set of $\{\lambda_i\}_{i\in\mathcal{I}}$, the distributed solution converges because: first, the objective function (9) is continuously differentiable and convex on the entire set of variables. Second, each step of $\mathbf{RS^l_n}$ is a minimization of (9) with respect to its own variable R_n, assuming others are held constant. Third, the optimal solution of each $\mathbf{RS^l_n}$ is uniquely attained, since its objective function is quadratic. The three conditions together ensure that the limit point of the sequence $\{R_n\}^{(t)}_{n\in\mathcal{N}}$, minimizes (9) for a given set of $\{\lambda_i\}_{i\in\mathcal{I}}$.

Second, we need to solve the master dual problem:

$$\begin{array}{ll} \text{maximize} & f(\lambda) \\ \text{subject to} & \lambda \ge 0 \end{array}$$

where $f(\lambda) = \max_R L(R,\lambda)$, which is solved in the first step. Since the solution to (9) is unique, the dual function $f(\lambda)$ is differentiable, which can be solved by the following gradient projection method:

$$\lambda_i \leftarrow \max\big\{0, \lambda_i + \theta\big((P_i-w_i)^2 - \varepsilon_i^2\big)\big\},\ \forall i$$

where $\theta > 0$ is a small positive step size. Condition (ii) guarantees the dual prices λ are updated in a larger timescale, as the dual algorithm requires.

The duality gap of $\mathbf{RS^g}$ is zero, and the solution to each $\mathbf{RS^l_n}$ is also unique. This finally guarantees that the equilibrium point of the decentralized algorithm is also the optimal solution of the global problem $\mathbf{RS^g}$. ■