Decentralized Server Selection Through Joint Proximity and Load Optimization

Patrick Wendell, Joe Wenjie Jiang, Jennifer Rexford, and Michael J. Freedman
Princeton University

ABSTRACT

With the advent of "cloud computing" and the growth of popular Web services, many networked services are replicated at multiple geographic locations. Such distributed services face the challenge of server selection — that is, directing an incoming client request to the appropriate server or data center, in the hope of reducing network latency or carefully tuning server loads. To meet these potentially conflicting goals, existing approaches use heuristics or rely on central coordination to perform server selection. In this work, we apply optimization theory to derive a simple, provably optimal, fully distributed solution to the server-selection problem. Our approach defines a global objective for a mapping service, and shows that decentralized mapping nodes performing small amounts of local computation and sharing limited information, can achieve the global objective. We also perform experiments, based on a 24-hour trace of a real operational CDN, that show that the distributed solution converges very quickly in practice.

1. INTRODUCTION

Many Internet services are replicated across multiple locations, to handle a high rate of requests while also minimizing network latency in reaching clients. Akamai [1] and other CDNs [14, 7] direct Web clients to HTTP proxies distributed across hundreds or even thousands of vantage points. Web search engines and other Web services offered by companies such as Google, Yahoo, Microsoft, and Amazon typically run at dozens of data centers spread all over the world. This trend of geographically-diverse server placement will only continue, especially with the growth of highly-interactive services such as the move of office applications into the cloud. These systems all face the challenge of replica selection—directing client requests to the appropriate service instance—based on both the location of the clients and the target load on the servers.

Today's services employ various techniques to direct each client to a specific service instance—whether an individual server or cluster of co-located servers. In DNS-based redirection, a DNS query to resolve a site's name (e.g., www.example.com) returns the IP address of a particular service instance; DNS queries from clients in different locations may return different responses. In HTTP-based redirection, Web servers return a special HTTP response message that directs the client to a different service instance (say, with a different server name). Some services apply other techniques, such as HTML rewriting, to present different clients with different URLs. Still, all of these techniques face a common underlying problem: how to match an incoming request—whether a DNS query from the client's local DNS server or an HTTP request from the client itself—to the appropriate service instance. In addition, these mapping decisions are often made by *multiple* nodes that are themselves distributed across multiple locations. As such, any practical server-selection solution should operate in a distributed fashion, preferably requiring only limited coordination amongst the nodes.

Various applications may have different notions of what constitutes a "good" choice of service instances, but the decision is usually a combination of two important factors:

Client proximity: Low network latency is especially important for small Web transfers or interactive applications (where network round-trip times can dominate performance), as well as large transfers (since TCP throughput is inversely proportional to round-trip time). Services commonly estimate latency between a client and potential replicas either using IP geolocation databases [11] or through network measurements [1, 15, 8, 9].

Server load: Server load also has a significant influence on user-perceived performance. Yet, cost may be an even more important reason for considering server load. Handling highly variable request loads requires excessive overprovisioning and runs afoul of common billing practices; for example, network providers often charge based on the 95th-percentile bandwidth usage over (say) all 30-minute periods in a month. In some sense, then, network costs can be considered in terms of server load. In any server-selection scheme, both request capacity and bandwidth cost will influence the preferred amount of load to imposed on a given replica. The value of both factors are likely to be heterogeneous from one replica to the next.

Thus, modern distributed services often apply heuristics that strike a balance between directing clients to the closest replica and achieving a target distribution of requests over the service instances. One common technique considers load as a binary metric, such that locality is first used to sort "non-overloaded" replicas, after which "overloaded" replicas are ordered by locality [8]. Other approaches rely on the DNS layer to direct each client to the nearest server, allowing the servers to shed excess load through HTTP-Redirection [5]. While certainly an improvement over simple round-robin techniques, these heuristics can be suboptimal and inefficient, and may lead to undesirable swings in server load or constant redirection of requests.

In this paper, we apply optimization theory to "derive" a provably-optimal, distributed solution to the server-selection problem. Given a target proportion of requests

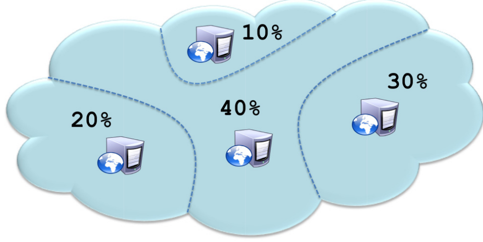

Figure 1: Partitioning client space onto servers

$\mathcal{N}$:	The set of mapping nodes
$\mathcal{C}$:	The set of clients
$\mathcal{I}$:	The set of server instances
R_{nci} :	The proportion of traffic load that is mapped to server i from client c by node n
α_{cn} :	Proportion of node n's traffic load from client c
s_n :	The proportion of total traffic load on node n
P'_i :	The desired proportion of overall traffic delivered to instance i

Table 1: Summary of key notations

to direct to each service instance, each node computes how to direct its share of the clients to the appropriate service instances. As an example, Figure 1 provides a hypothetical partitioning of requests from different geographic locations to four replica instances, visualized in two-dimensional space, to achieve a target 40/30/20/10 splitting ratio. The desired ratio may be influenced by server capacity, bandwith cost, or other factors. The areas around each service instance represent the clients which are directed to it. Our distributed solution is highly efficient, requiring only minimal exchange of information between the mapping nodes. We prove that our algorithm converges to an optimal solution and experimentally verify that the system converges quickly—often in just a few iterations.

The remainder of this paper is organized as follows: Section 2 quantifies "good" server selection as a global optimization problem involving both client proximity and server loads. Section 3 introduces a decentralized algorithm for finding this optimum, whereby separate mapping nodes communicate with one another to achieve globally optimal behavior. Section 4 describes encouraging simulation results based on an initial implementation of this algorithm, and Section 5 describes how our method relates to existing approaches. Finally, Section 6 concludes.

2. SERVER-SELECTION PROBLEM

This section characterizes the server-selection problem in terms of a global optimization. We present a network model to describe a distributed service and introduce a method which strikes an optimal balance between locality and targeted server load distribution. We also introduce some notation used in this paper, summarized in Table 1.

2.1 Network Model and Mapping Service

Consider a network that provides a server mapping service. Denote $\mathcal{C}$ as the set of clients that require mapping, $\mathcal{I}$ as the set of server instances, and $\mathcal{N}$ as the set of mapping nodes. Each replica $i \in \mathcal{I}$ may represent a single server or a cluster of servers such as a large datacenter. We use the terms instance, server, and replica interchangeably throughout the remainder of this paper.

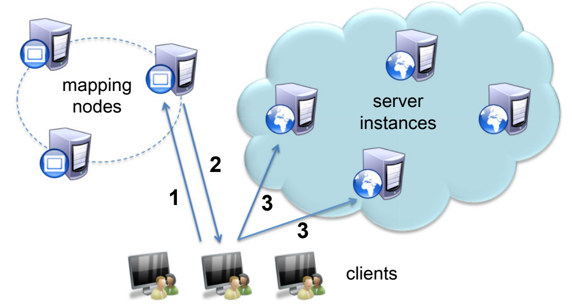

Figure 2: Server selection with mapping nodes

Figure 2 illustrates the stages of a mapping service. A mapping node $n \in \mathcal{N}$ receives the request from a client c (step 1). The node maps the client to a content instance $i \in \mathcal{I}$ and returns the result to the client (step 2). In practice, each client $c \in \mathcal{C}$ can represent a group of aggregated end hosts, *e.g.*, according to their zip codes. Therefore, it is possible to direct the load from one client to one or multiple server instances (step 3). Let $R_{nci} \in [0, 1]$ denote the proportion of traffic load that is mapped to server i from client c by node n, *i.e.*, $\sum_i R_{nci} = 1$, for $\forall(c, n)$.

Each node n observes its total traffic load, *i.e.*, the number of requests, from its clients. We normalize the workload as the proportion of total load over all service nodes and denote it as as s_n. Let $\alpha_{cn} \in [0, 1]$ denote the proportion of n's traffic load from client c, *i.e.*, $\sum_c \alpha_{cn} = 1$, $\forall n$; in particular, $\alpha_{cn} = 0$ when client c is not served by n.

The decision variable for each mapping node n is R_{nci}, $\forall c \in \mathcal{C}$, $\forall i \in \mathcal{I}$, *i.e.*, how much of client c's traffic, should node n direct to service instance i. Other information, such as s_n and α_{cn} are assumed as constant problem parameters. We will revisit this assumption and discuss the application context in practice. All the above information is local, maintained by node n individually.

2.2 Locality-Aware Server Selection

One of the server selection goals is to minimize network latency. To quantify the distance between a client and an instance, we define a distance function $dist(c, i) \in [0, 1]$. This function measures the latency cost of pairing client c with instance i, which can be defined using geographical proximity, network coordinates, or other distance estimation methods. The $dist(c, i)$ information is calculated locally by node n for all clients under its service, and it remains a constant in our problem.

The choice of $dist(c, i)$ function implies our preference over different notions of the global proximity. For instance, if we want to minimize the average network latency, the *dist* functions can simply be RTTs or Euclidean distances. Alternatively, if we want to avoid long-delay connections, *e.g.*, minimizing the maximum latency or percentile-based latencies, we can choose *dist* with increasing marginal penalty as distance grows. For example, in our experiments, we chose a translated logistic function as shown in Figure 3, which implies that a range of latencies below a certain threshold are acceptable, yet very large latencies are not. The flexibility in defining $dist(c, i)$ allows us to adhere to a single frame-

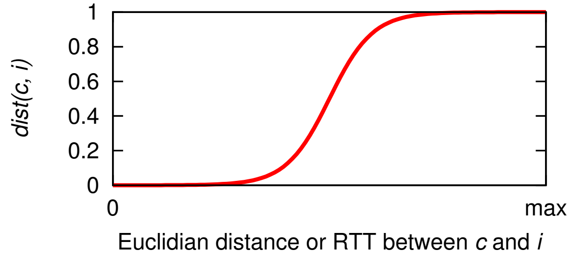

Figure 3: *dist(c, i)* per Euclidean distance/RTT

work while being open to different notions of "optimality" in network latency.

To achieve locality-aware server selection, we define a global system goal, *i.e.*, the global proximity function that reflects the overall user latency:

$$prox^g = \sum_{n \in \mathcal{N}} s_n \sum_{c \in \mathcal{C}} \alpha_{cn} \sum_{i \in \mathcal{I}} R_{nci} \cdot dist(c, i) \quad (1)$$

by summing across the distance between all client-instance pairings. By minimizing $prox^g$, we map clients to closer server instances and reduce end-user latencies.

2.3 Load-Aware Server Selection

The second consideration in server selection is to direct requests to different server instances, such that each instance operates at a targeted workload. Denote $P'_i \in [0, 1]$ as the desired proportion of traffic assigned to instance i, which can be configured by the service provider to reflect its preferences. Also let P_i be the true proportion of traffic assigned to instance i, which is realized by a given set of R_{nci}'s as:

$$P_i = \sum_{n \in \mathcal{N}} s_n \sum_{c \in \mathcal{C}} \alpha_{cn} \cdot R_{nci} \quad (2)$$

We hope each server operates under the desired workload so that the server is not overloaded or under-utilized, *e.g.*, $|P_i - P'_i| < \epsilon$. To achieve this, we introduce a penalty function $cost(P_i - P'_i)$ that measures the cost of deviating from the desired server load. We use the square of the deviation from desired server load in order to penalize both over and under-provisioned servers.

$$cost(P'_i - P_i) = (P'_i - P_i)^2 \quad (3)$$

Summing up all instances gives the global load deviation cost, $cost^g$:

$$cost^g = \sum_{i \in \mathcal{I}} cost(P'_i - P_i) \quad (4)$$

2.4 Jointly Optimizing Proximity and Load

We formulate the locality-and-load aware server selection problem as the following optimization problem SS^g:

$$\begin{aligned} \text{minimize} \quad & prox^g + \beta \cdot cost^g \quad (5) \\ \text{subject to} \quad & \sum_{i \in \mathcal{I}} R_{nci} = 1 \ \forall (n, c) \\ \text{variables} \quad & R_{nci} \geq 0, \ \forall (n, c, i) \end{aligned}$$

where β reflects the weight of the service provider's preference to load-awareness. Smaller values of β lead to better locality, at the expense of poorer load balancing.

The server-selection problem (5) accepts a unique solution under assumptions that $cost(\cdot)$ is convex [4]. In this work, we choose the quadratic cost function in Eq. (3). Since the *prox* function is linear in R_{cni}, we can efficiently solve the problem using quadratic programming.

However, solving the problem in a centralized way would require the coordination of all mapping nodes and the sharing of per-client, per-node information. A central authority is needed to perform all the computation and pass the solution to each node. The total amount of message passing is proportional to $|\mathcal{N}| \times |\mathcal{C}| \times |\mathcal{I}|$, and the computation repeats for every change in the problem inputs. Such a centralized approach could be unacceptably expensive and suffer from a single point of failure. This motivates us to look for a distributed solution, such that each mapping node only needs to know its local information.

3. DECENTRALIZED SELECTION

In this section, we propose a decentralized solution to the locality-and-load aware server selection problem. We first propose an optimization decomposition to solve the global problem in a distributed manner. We then translate the mathematical solution to an iterative algorithm and discuss its protocol implementation.

3.1 Decomposing the Optimization Problem

We next show how to decompose the global server-selection problem into multiple local server-selection problems while guaranteeing that the solution converges to the global optimum. The distributed solution does not require sharing of information proportional to the entire client space. Instead, each node performs a smaller local optimization based on its own view of the client space.

Consider the global proximity function $prox^g$ (1), consisting of the local proximity contributed by each node:

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

where

$$prox^l_n = s_n \sum_{c \in \mathcal{C}} \alpha_{cn} \sum_{i \in \mathcal{I}} R_{nci} \cdot dist(c, i) \quad (6)$$

Also consider the global cost function $cost^g$ given in Eq. (4). The true proportion of traffic assigned to instance i can be rewritten 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 $P_{ni} = s_n \sum_{c \in \mathcal{C}} \alpha_{cn} \cdot R_{nci}$ denotes the traffic load contributed by node n on instance i. We also use P_{-ni} for the traffic load contributed by nodes other than n, which is independent of node n's decisions.

The local version of server selection for node n can be defined as the following optimization problem SS^l_n:

$$\begin{aligned} \min \quad & prox^l_n + \beta \sum_{i \in \mathcal{I}} cost(P'_i - P_{ni} - P_{-ni}) \quad (7) \\ \text{s.t.} \quad & \sum_{i \in \mathcal{I}} R_{nci} = 1, \ \forall c \\ \text{var} \quad & R_{nci} \geq 0, \ \forall (c, i) \end{aligned}$$

In this problem, each node only needs to know its local info (s_n, α_{cn}), and the aggregated info P_{-ni} from other nodes. The following theorem shows that nodes performing *local* server selections, provided information is exchanged in an appropriate order, leads to the *global* optimum.

Initialization. For each node n:

(1) Initializes an arbitrary solution $\{R_{nci}\}_{c\in\mathcal{C},i\in\mathcal{I}}$.
(2) Computes its $\{P_{ni}\}_{i\in\mathcal{I}}$.
(3) Passes $\{P_{ni}\}_{i\in\mathcal{I}}$ to all other nodes n'.

Iteration. For each node n:

(1) Receives the latest $\{P_{n'i}\}_{i\in\mathcal{I}}$ from other n'.
(2) Waits until node $1, 2, \ldots, n-1$ finishes optimizing their local server selection.
(3) Computes P_{-ni} based on received information.
(4) Solves SS_n^l and computes $\{P_{ni}\}_{i\in\mathcal{I}}$.
(5) Passes $\{P_{ni}\}_{i\in\mathcal{I}}$ to all other nodes n'.
(6) Stops if $\{P_{n'i}\}_{i\in\mathcal{I}}$ from other n' do not change.

Table 2: Decentralized server selection

Theorem 1. *Each node iteratively optimizing the local server selection SS_n^l (7), for $n = 1, 2, \ldots, \mathcal{N}$, based on the updated information P_{-ni} from other nodes, i.e.,*

$$R_{nci}^{(t+1)} = \operatorname{argmin} SS_n^l(\ldots, R_{n-1ci}^{(t+1)}, R_{nci}, R_{n+1ci}^{(t)}, \ldots)$$

leads to a global optimum of server selection, SS^g (5).

Proof: Our distributed solution implements the nonlinear Gauss-Seidel algorithm, which guarantees convergence if the following conditions are satisfied (per [3, Ch. 3, Prop. 3.9], [2, Prop. 2.7.1]): First, the global objective function must be continuously differentiable and convex on the entire set of variables. Our global function SS^g (5) fulfills the condition on its variables $\{R_{nci}\}_{c\in\mathcal{C},i\in\mathcal{I}}, n \in \mathcal{N}$. Second, each step of the local optimization must minimize the global objective with respect to a subset of the global variables, assuming others are held constant. Our local problem, SS_n^l (6), is a minimization of the global objective SS^g (5), with respect to the local variables $\{R_{nci}\}_{c\in\mathcal{C},i\in\mathcal{I}}$, given fixed decisions from other nodes. Third, the optimal solution of each local problem must be uniquely attained. Since SS_n^l is strictly convex on the local variable $\{R_{nci}\}_{c\in\mathcal{C},i\in\mathcal{I}}$, this third condition is met. The three conditions together ensure the convergence property of our distributed solution, *i.e.*, the limit point of the sequence $\{R_{nci}\}_{\forall(c,i)}^{(t)}$, $n = 1, 2, \ldots, |\mathcal{N}|$, minimizes SS^g over the entire variable space. ∎

3.2 Distributed Algorithm

After showing the convergence property of the decentralized server selection, we next formally present the distributed server selection algorithm in Table 2.

The algorithm shown in Table 2 updates each node's decision in a sequential manner. At the initialization stage, each node picks an arbitrary solution, *e.g.*, one that minimizes local proximity only. It must choose a solution which is locally attainable, since no communication has taken place. Each node computes and passes the aggregated load information $\{P_{ni}\}_{i\in\mathcal{I}}$ to other nodes. At the iteration stage, one node solves its local server selection, based on the latest information received from other nodes. After finishing its local optimization, each node passes the updated information $\{P_{ni}\}_{i\in\mathcal{I}}$ to other nodes.

The decentralized server selection reduces both the amount of data shared between the nodes and the computational complexity of each calculation. The local problem is of size $|\mathcal{C}| \times |\mathcal{I}|$, which is much smaller than the global problem. The decrease in message passing is discussed next.

3.3 System Architecture

Any implementation of our server-selection algorithm must address certain pragmatic design issues. Here we discuss two major considerations: (1) the choice of techniques for message passing and (2) assumptions about the degree to which system inputs change.

3.3.1 Inter-Node Communication

Our algorithm requires communication between nodes to share updated local decisions. Here, we briefly discuss three popular methods for sharing server state and their implications for distributed server selection.

Ring: Nodes are arranged in a ring (as in consistent hashing), and each node transmits $O(|\mathcal{I}|)$ information to its successor node. This method has the merit of reducing the amount of message passing, *i.e.*, the $\{P_i\}_{i\in\mathcal{I}}$ information can be incrementally updated, without the need to broadcast local P_{ni} information to all nodes. Thus, the total communication complexity is $O(|\mathcal{N}||\mathcal{I}|)$. On the other hand, information is slow to propagate—linear in $|\mathcal{N}|$—and failures can further interrupt propagation.

Broadcast: Each node broadcasts the local $\{P_{ni}\}_{\forall i\in\mathcal{I}}$ message to all other nodes asynchronously. The P_i is reconstructed using the latest information received from other nodes. Nodes update their decisions in a parallel fashion, which does not require any coordination and responds more quickly. The total communication complexity is $O(|\mathcal{N}|^2|\mathcal{I}|)$. While such parallel update does not guarantee the convergence to the global optimum, in practice, the resulting solution is often close to the optimum. However, such asynchronous update would result in higher fluctuations of the objective value during the transient period.

Gossiping: Each node propagates its local information to a subset of nodes using a gossip protocol, and local server selection happens in an asynchronous manner. The gossiping method is potentially applicable to a large number of nodes, by achieving less message passing than broadcast methods, yet still propagating in a logarithmic (in $|\mathcal{N}|$) number of transmission rounds.

3.3.2 System Dynamics

Our optimization framework solves a joint locality-and-load optimization assuming the problem is static. In practice, the system is dynamic in many ways. First, the client request rates (s_n, α_{cn}) can vary from time to time. Second, the preferred server load P_i' can change due to server capacity re-planning or reservation for other purposes. Third, a mapping node or server may go up and down. Any change defines a new problem and a new optimal solution.

Our distributed algorithm is tolerant to such system dynamics by migrating from one set of solutions to another. Since the computation is done locally, system changes can propagate through inter-node communications. Though the server-selection decisions, hence the locality and server loads, are non-optimal during the transient period, they eventually converge to the new optimum.

For example, when a new instance is added, the P_i will be initially set zero, and updated to all mapping nodes. The distributed algorithm directs the client requests to the new server. After several iterations, the new server would end up receiving the optimal amount of traffic. As long as any major change is relatively infrequent compared to the convergence speed (which will be shown in the next section),

our system operates in a stable region, and can quickly react and converge to its optimum point.

4. EXPERIMENTAL RESULTS

In this section, we use trace data to test the assumptions required for convergence. We also measure the performance of the algorithm in simulated topologies.

4.1 Analysis of Typical Server Workload

To fully understand the workload experienced by a distributed mapping service, we analyzed DNS log files from CoralCDN, a popular content distribution network which uses DNS to perform server selection [7]. Our dataset consisted 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 [11].

Stability of Request Rate: Our problem definition does not specify the time interval for calculating request rates. For load balancing purposes, a finer granularity is preferred, as request volume can be guaranteed within very short time periods. However, under very fine granularities client request rate becomes increasingly unpredictable. We select an interval of 10 minutes and show that client request rates are quite predictable under this interval.

To test the regularity of request rate, we chose four aggregation mechanisms, IP prefix, U.S. Zip Code, U.S. Area Code, and U.S. State.[1] U.S. requests accounted for 32.4% of overall traffic during this time. We predict request volume in a given 10-minute interval based on an exponentially weighted average of the previous intervals. We then record the percentage by which the true data deviates from our prediction. Figure 4 (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 averaged difference over a 2-hour interval for one client. Figure 4 (bottom) is a CDF of all traffic from these same clients. Fortunately, 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, such as state, leads to even better request stability, but at the cost of locality precision. A tradeoff emerges between precisely identifying client locale and ensuring that request rates per client group are predictably stable.

4.2 Simulating the Distributed Algorithm

To demonstrate the convergence of our algorithm under realistic conditions, we ran simulations using the CoralCDN trace data. Our simulation consisted of 10 randomly placed mapping nodes and four service instances. Each client communicates with the nearest mapping node. We use a desired split of 40/30/20/10 (as in Figure 1).

The optimization at each node is performed using IPOPT [13], a software library capable of solving nonlinear optimization problems in polynomial time.[2] Our mapping nodes aggregate clients by area code, and the runtime of each local optimization was around 200 milliseconds, running on a 1.8 GHz dual core machine.

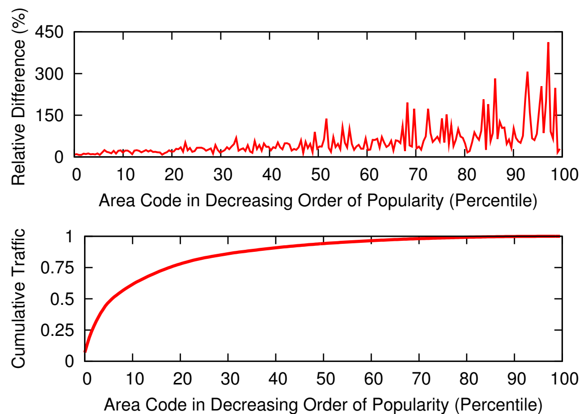

Figure 4: Stability of area code request rates

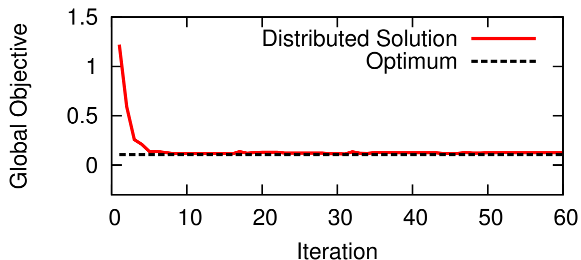

Figure 5: Convergence towards global optimum

Convergence Speed: Figure 5 demonstrates the convergence speed of our distributed algorithm towards the optimal objective (5). Each iteration represents a local optimization by a single node. Convergence is achieved after the first ten iterations, *i.e.*, all nodes have completed one pass of their local optimizations, which is around 1.87 seconds in this experiment. In practice, the computation time plus the communication time for each iteration should take less than a handful of seconds, thus, achieving the global convergence in tens of seconds.

Server Load Distribution: To see how well our algorithm distributes load, we chose large β to give more weight on load distribution over locality. We also introduce changes in various inputs to observe behavior in a dynamic setting. Figure 6 shows that server loads quickly converge to the desired levels both initially and after the problem input changes. Phase A shows servers converging from a random initial point to a requested split of 40/30/20/10. In phase B, we adjust the requested split to 70/15/10/5, holding request rate constant. In phase C, we randomly adjust the volume of traffic per client by up to 100%. Finally, in phase D we change the requested split to 25/25/25/25 and alter client request behavior as in phase C. In every phase, the request loads converge to (and remain) within 3% of requested proportions by the phase's 10th iteration. In phase C, each server deviates less than 15% from requested proportions during the re-convergence. The consistent speed of convergence offers a major benefit to service providers, as no server

[1] These mechanisms allow us to study the exact same set of clients under different levels of aggregation. In practice, an algorithm would use similar regional clustering for other countries.

[2] IPOPT applies to general convex cost functions. We can scale to a very large number of clients and server instances, *e.g.*, tens of thousands of variables, if the cost function is quadratic and applying a quadratic programming solver.

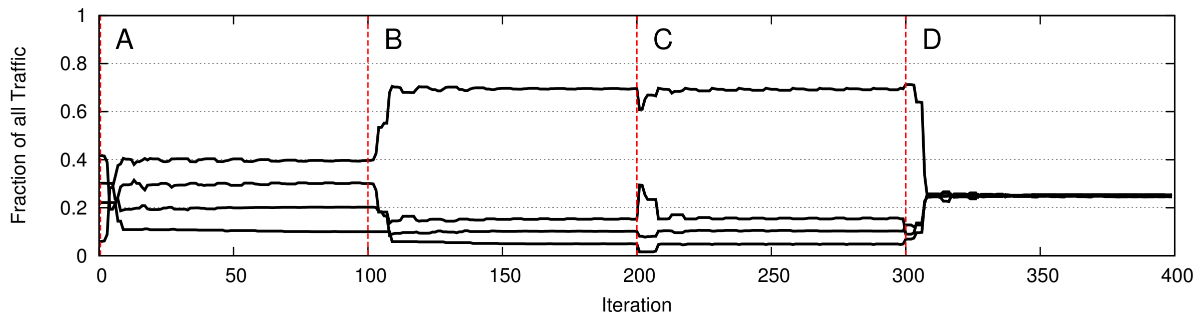

Figure 6: Convergence of server load under input changes

instance remains over or under loaded for very long, despite fluctuations in client behavior or desired server load.

Improved Local Proximity: While we selected high β to favor server load, mapping nodes achieved low local proximity as well. We define an "ideal" proximity measurement for each node as the lowest attainable proximity, *i.e.*, each client is always mapped to the server with the least distance cost. This value acts as a baseline from which to compare. Nodes averaged 7% (stdev 3.3%) above optimal proximity.

Larger Scale Simulations: To test the algorithm's effectiveness of balancing load across many servers, we created a topology with 100 servers, each demanding 1% of traffic. Convergence remained fast, with all servers falling within 10% of desired traffic load after ten iterations. After 50 iterations, the average server had 1.000% of load with a standard deviation of .001766.[3] Local proximity as measured by mapping nodes was 4% above optimal (stdev 2.4%).

5. RELATED WORK

The previous section demonstrates that a distributed mapping system can jointly optimize for proximity and load by leveraging the stability of aggregate request rates. Generally, there are two categories of approaches with similar goals: (1) distributed techniques that provide real-time relief to overloaded replicas [6, 5] based on heuristics that adaptively react to server feedback (but at the risk of sub-optimality and instability) and (2) centralized mechanisms [10] that do not scale well and require complex replication for reliability. In this sense, our work is a major improvement: a solution both provably optimal and capable of running in a completely decentralized manner.

Feedback-based approaches do address certain gaps in our model. First, there is the possibility of bursty requests within our chosen interval, potentially overloading servers. Second, request volume may not directly translate to CPU load. DNS-based techniques further muddle this translation by assuming uniform client load behind each resolver. These issues are more pronounced when the mapping service runs on a large number of small servers. Yet, the growing trend of deploying services across a handful of large data centers, can potentially alleviate these problems. The ability of larger datacenters to absorb bursty requests and better tolerate load variability makes our approach more amenable to these increasingly popular infrastructures.

6. SUMMARY AND FUTURE WORK

This paper motivates the need for optimization-based server selection in replicated Web services. We show that, under the assumption of a stable request rates per geographic region, one can construct an optimal mapping service by solving a joint proximity-and-load optimization problem. We introduce a decentralized algorithm for arriving at this optimum, without requiring global sharing of information about client request rates.

In future work, we plan to study further dynamic scenarios, including both sudden changes of client distributions (flash crowds) and desired server loads. We also seek to quantify the performance gap, both theoretically and experimentally, between various heuristic approaches and our solution. Finally, we are developing a prototype implementation of this algorithm in a new distributed DNS service [12].

[3] It should be noted that the uncertainty in client request rate under a more dynamic model will cause this variance to increase.

7. REFERENCES

[1] Akamai Technologies. http://www.akamai.com/, 2009.
[2] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.
[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.
[4] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
[5] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic load balancing for scalable distributed web systems. In *MASCOTS*, Aug. 2000.
[6] M. Colajanni, P. S. Yu, and D. M. Dias. Scheduling algorithms for distributed web servers. In *International Conference on Distributed Computing Systems (ICDCS)*, page 169, 1997.
[7] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, Mar. 2004.
[8] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *NSDI*, May 2006.
[9] 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.
[10] M. Pathan, C. Vecchiola, and R. Buyya. Load and proximity aware request-redirection for dynamic load distribution in peering CDNs. In *OTM*, Nov. 2008.
[11] Quova. http://www.quova.com/, 2009.
[12] A. Schran, J. Rexford, and M. J. Freedman. Namecast: A reliable, flexible, scalable DNS hosting system. Technical Report TR-850-09, Princeton University, Apr. 2009.
[13] A. Wachter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106:25–57, 2006.
[14] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *OSDI*, Dec 2002.
[15] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *SIGCOMM*, Aug. 2005.