Optimal Resource Utilization in Content Distribution Networks

Yee Jiun Song Venugopalan Ramasubramanian Emin Gün Sirer
Dept. of Computer Science, Cornell University, Ithaca, NY 14853
{yeejiun, ramasv, egs}@cs.cornell.edu

Abstract

This paper examines replication in content distribution networks and proposes a novel mechanism for optimally resolving performance versus cost tradeoffs. The key insight behind our work is to formally and analytically capture the relationship between performance, bandwidth overhead and storage requirements for a web cache, express the system goals as a mathematical optimization problem, and solve for the optimal extent of replication that achieves the desired system goals with minimal overhead. We describe the design and implementation of a new content distribution network based on this concept, called CobWeb. CobWeb can achieve a target lookup latency while minimizing network and storage overhead, minimize access time while keeping bandwidth usage below a set limit, and alleviate "flash crowd" effects by rapidly replicating popular objects through fast and highly adaptive replica management. We outline the architecture of the CobWeb system, describe its novel optimization algorithm for intelligent resource allocation, and compare, through simulations and a physical deployment on PlanetLab, CobWeb's informed, analysis-driven replication strategy to existing approaches based on passive caching and heuristics.

1 Introduction

Caching can significantly improve user perceived latencies as well as reduce the amount of aggregate network traffic. The popularity of the web makes caching a natural place to apply caching techniques to improve client performance, reduce server load, and minimize network traffic. Web caches to date have been deployed in two different settings, one driven by clients and one by content providers.

Web caches that are placed close to the clients are commonly known as proxy caches. Such demand-side caches exploit temporal locality within the clickstream of a single user as well as spatial locality stemming from the common interests of independent users. Proxy caches depend on passive monitoring and opportunistic caching, where each proxy only caches objects that have been requested by a client that is directly connected to it. Passive opportunistic caching severely limits potential benefits because web traffic is well-known to follow a Zipf distribution, with a heavy tail [3, 7, 1]. Since the heavy tail of the distribution limits spatial locality, past work has examined cooperative web caching, aimed at aggregating the cache contents of multiple web proxies to obtain greater caching benefits. Cooperative caching schemes that have been proposed include hierarchical [6, 30], hash-based [16, 24], directory based [9, 20, 28], and multi-cast schemes [21]. Yet past work on cooperative caching has examined only passive mechanisms for cache control, and an interesting negative result has demonstrated that cooperative caching provides performance benefits only within limited population bounds [31]. The large heavy-tail of the popularity distribution, combined with purely passive measures for cache control, makes it difficult to achieve high cache hit ratios.

Web caches can also be placed within the network to aid content distribution. In particular, companies such as Akamai provide content distribution services to web site operators by placing servers in strategic locations to cache and replicate content. Such networks of servers are commonly known as *content distribution networks* (CDNs), and are driven by content providers rather than content consumers. In contrast to the demand-driven nature of web proxies, most CDNs proactively replicate web objects throughout the network using heuristics aimed at load balancing and improving performance [10, 29]. These heuristics aim to maximize the effective benefit from the bandwidth spent on proactive content distribution, but typically do not provide any hard performance guarantees.

The fundamental challenge faced by any web cache is to decide which objects to replicate and to what extent. Proxy web caches sidestep this problem by passively caching objects that local clients have requested. In doing so, they limit the benefits that can be realized through caching to only those objects that have been fetched by the client population. CDNs, on the other hand, utilize heuristics which offer little control over the performance characteristics and resource consumption of the resulting system. For example, there is no way to guarantee a certain hit rate in such systems, or to cap bandwidth consumption at a desired limit.

In this paper, we describe a novel, principled approach for determining which objects to cache and to what extent in a distributed CDN. We analytically model the costs and performance benefits of replication, formalize the tradeoffs as an optimization problem, and use a novel numerical solver to find a near-optimal solution that maximizes global system goals, such as achieving a targeted hit rate, while respecting resource limits, such as bandwidth consumption. Our system, CobWeb, is a global network of caching proxies that uses this analysis-driven approach, which utilizes the popularity, size, and update rate of web objects to compute the replication strategy. The resultant solution provides low latency lookup to clients while minimizing the storage and network overhead incurred by CobWeb proxies.

Analytically modeling the overhead costs and performance benefits of replication enables CobWeb to convert this systems problem to an optimization problem. The optimization problem can then be solved to provide, for instance, minimal lookup latency while staying within a network bandwidth budget, or to achieve a targeted lookup performance while minimizing bandwidth consumption. This enables CobWeb to offer highly adjustable performance characteristics that is not available in heuristics-based systems.

Through simulations and measurements from a real world deployment, we make a case for structured, analysis-driven web caching over opportunistic heuristic-driven caching. We show that our system provides high performance and low overhead when compared to passive caching systems, and propose deployment strategies for integrating our system into the Internet.

The rest of this paper is structured as follows. In the next section, we describe the analysis-driven replication technique that enables CobWeb to resolve the performance-overhead tradeoff encountered in web caching. In Section 3, we outline the overall architecture of the CobWeb cache. Section 4 describes the current CobWeb implementation. In Section 5, we evaluate the performance of CobWeb through extensive simulations and a physical deployment on PlanetLab, and compare it to existing CDNs as well as passive caching. Section 6 describes related work and Section 7 summarizes our contributions.

2 Resource Optimal Replication

The central insight behind CobWeb is that the fundamental tradeoff between performance and the cost required to achieve that performance can be treated as an optimization problem. CobWeb analytically models this tradeoff, poses it as an optimization problem, and finds the optimal replica placement strategy. This optimization analysis enables CobWeb to make informed decisions during replication in order to meet performance expectations with minimal cost. Conversely, this analysis can be used to optimize performance while keeping network and storage consumption below a fixed limit.

CobWeb takes advantage of structured organization of the system to analytically model resource-performance tradeoffs. Several structured overlay systems, which organize the network to form well-defined topologies with regular node degree and bounded diameter, have been proposed in the recent past [23, 27, 19, 25, 12, 33, 18]. These systems called Distributed Hash Tables (DHTs) provide high failure resilience and scalability through decentralization and self-organization. By layering CobWeb on a DHT we not only inherit its high failure resilience and scalability, but also leverage its regular topology to concisely capture performance-overhead tradeoffs. We illustrate this structured analysis using Pastry [25] as an example overlay.

Pastry organizes the network as a ring by assigning identifiers to nodes from a circular identifier space. Objects are also assigned an identifier from the same space and stored at the node with the closest identifier, called the *home node*. When queries are injected into the system, Pastry routes the queries towards the home node by successively matching prefix digits in the identifier of the queried object. This routing process is aided by long distance contacts with different numbers of matching prefix digits and takes $O(logN)$ hops in a network of N nodes.

The structured organization provides an opportunity for replication to shorten the route of the lookup path. By replicating objects at all nodes that are within i hops from the home-node, the lookup latency can be reduced to $\log(N) - i$ hops. We formalize this concept by defining a *replication level* for each object. An object is said to be replicated at level l if it is stored at all nodes in the system with l matching prefix digits. An l level object has lookup latency of l hops and is replicated at $\frac{N}{b^l}$ nodes in the system. Figure 1 illustrates the concept of replication levels in Pastry.

Structured replication of this manner enables CobWeb to concisely express the replication cost and lookup latency for each object. CobWeb extends this to analytically frame the global performance-overhead tradeoffs.

2.1 Analytical Model

We pose performance-overhead tradeoffs in a system of M objects through optimization problems of the following form:

$$\text{Min. } \sum_{1}^{M} c_m(l_m) \quad \text{s.t. } \textstyle\sum_1^M p_m(l_m) \leq \quad T \qquad (1)$$

In the above expression, l_m represents the replication level of object m and functions $c_m(l)$ and $p_m(l)$ represent

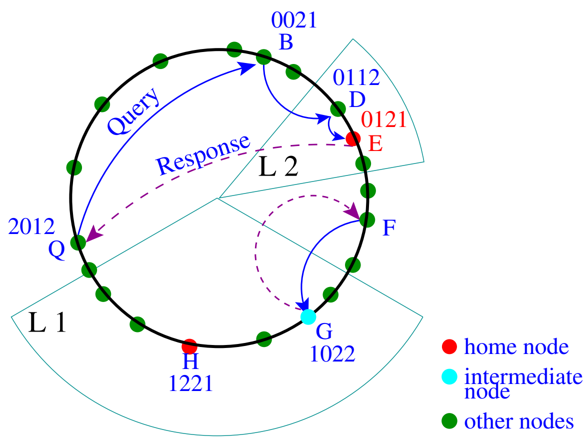

Figure 1: **CobWeb Replication on top of Pastry.**

cost and performance tradeoffs respectively for each object. The above expression poses an optimization problem to minimize the cost required to achieve a performance target T. The converse problem to minimize lookup latency without exceeding a bound on cost is of a similar form and represented by swapping the positions of cost and performance tradeoff functions.

We model the performance of the system with the average lookup latency of queries since content distribution networks are primarily concerned with providing users with low latencies through a high hit rate. The performance tradeoff for an object at level l can be modeled as $p_m(l) = q_m D(l)$, where q_m is the popularity of the object in terms of the fraction of total queries it receives, and $D(l)$ is the network latency in the underlying DHT to traverse l hops. For a DHT that does not take into account network proximity during self-organization the average delay is the same at all levels and $D(l) = l$. However, for DHTs that self-organize based on network proximity [5], the latency $D(l) = \sum_0^l d_j$, that is the sum of the average latencies at each hop until l. The values of d_j can be experimentally determined in a deployed system and used during optimization to account for the latency differences across different levels.

The cost tradeoff for an object can be modeled based on the goals of the system. If the concern is storage cost, the cost function for object m is $c_m(l) = s_m \frac{N}{b^l}$, where s_m is the size of the object. Given that disk storage is cheap and disk capacity is rapidly increasing, storage cost is unlikely to be of much interest in practice. Network bandwidth, on the other hand, is expensive and often the bottleneck in distributing large objects. Hence we take special care to model the network bandwidth cost accurately.

The bandwidth consumption for CobWeb consists of three components: the update cost required to keep all replicas up to date, replication cost to push copies of the object, and any maintenance overhead required by CobWeb to manage replication. The cost to update a single copy of an object m is $s_m u_m$, where s_m is the size of the object and u_m is the update rate of the object, that is, the number of updates seen by the object in unit time. The management cost in CobWeb is a constant A for each replica in the system.

The replication cost of an object depends upon the current level of replication of the object. To increase the extent to which an object is replicated, that is, to reduce its replication level, a bandwidth cost is incurred to create additional replicas. Increasing an object's replication level (reducing the amount it is replicated), on the other hand, can be accomplished at no cost. The replication cost, $r_m(l)$ for an object can be represented thus:

$$r_m(l) \quad = \quad \begin{cases} s_m \left(\frac{N}{b^l} - \frac{N}{b^{j_m}}\right) & \forall\, l < j_m \\ 0 & \forall\, l \geq j_m \end{cases}$$

Here, j_m is the current replication level of object m. The overall total network overhead can be expressed as:

$$c_m(l) = (A + s_m u_m)\frac{N}{b^l} + r_m(l)$$

Using the above cost and performance functions, computing the optimal replica placement strategy involves computing a vector $L = \{l_1, l_2, \cdots, l_m\}$ such that the cost and performance functions satisfy a desired criteria. Given the size, update rates, and popularity of objects, CobWeb computes the optimal replica strategy in two possible configurations. In the first configuration, we set a target lookup latency, T_L, and compute the replica placement strategy that will achieve this target with the minimum cost. In this configuration, CobWeb provides a knob that allows system administrators to tune the performance of the system. For example, a target of 0.5 ensures that at least 50% of all queries do not require a network hop. That is to say, the system will guarantee a hit rate of at least 50%. In the second configuration, CobWeb minimizes the average lookup latency of queries subject to a limit on resource consumption. Given that our measure for cost is bandwidth overhead, a system administrator can set the amount of bandwidth, T_B that CobWeb can consume over a time interval. CobWeb then computes the replica placement strategy that will produce the best lookup performance within these limits.

The above optimization problems are NP-Hard because the replication levels of objects take integral values. One approach to perform efficient optimization efficiently is to model the parameters analytically and obtain a closed-form solution through mathematical derivation. Beehive [22], a replication framework of similar flavor to CobWeb, uses such an approach. Beehive obtains analytical solutions by assuming that popularity follows a Zipf distribution and objects are homogeneous in size and update rates.

However, Beehive's simple analysis-driven techniques cannot solve the optimization problems that arise in CobWeb for three fundamental reasons. First, Web objects have orders of magnitude differences in their size and update rates [8]; sizes can range from a few kilobytes to several megabytes and update intervals from a few seconds to no updates whatsoever. Beehive often ends up replicating a large or frequently updated object to a greater extent than a small static object of slightly less popularity. Consequently, Beehive can consume significantly more bandwidth than necessary. Second, Beehive optimizes for storage cost and cannot handle the precise estimate of bandwidth consumption modeled earlier. Finally, even though web objects are known to satisfy Zipf popularity distribution, sudden increases in object popularities during flash-crowds can cause deviations from the Zipf behavior and render Beehive's solution sub-optimal.

2.2 Numerical Optimization

CobWeb employs fast and accurate decentralized numerical techniques to solve the above optimization problems. These techniques, consisting of an optimization algorithm and a distributed tradeoff aggregation mechanism are part of a module we developed called Honeycomb. Honeycomb provides an $O(M \log M \log N)$ algorithm that resolves the general performance-overhead tradeoff problem in expression 1 and optimizes the overhead to the granularity of one object. The only assumption made by Honeycomb is that $c_m(l)$ and $p_m(l)$ are monotonic in l for each object. Monotonicity implies that the optimal solution lies at the boundaries of the constraint.

Honeycomb achieves high accuracy to the granularity of one object by finding upper and lower bounds for the optimal cost differing in replication levels for at the most one object. These upper and lower bounds are exact optimal solutions to problem 1 with slightly different constraints; one with a constraint $T_1 \leq T$ and another with constraint $T_2 \geq T$. The solutions L_1^* and L_2^* differ in at the most one object, that is, there may be one object that has a different replication level in L_1^* and in L_2^*. Note that the optimal solution L^* for the original problem 1 with constraint T may actually decide to replicate objects differently from L_1^* and L_2^*.

The small deviation in our solution compared to the true optimal has little impact on the performance of our system. CobWeb ends up replicating at the most one object per node more than the optimal. Given that an Internet scale content distribution system hosts millions of an object, this deviation is tiny and almost negligible. Hence, we avoid using expensive optimization techniques such as branch-and-bound or prune-tree search to find the true optimal.

Honeycomb determines the upper and lower bound solutions, L_1^* and L_2^*, through the use of a Lagrange multiplier to transform the constrained optimization problem to minimize $f(L, \lambda) = \sum_1^M c_m(l_m) - \lambda(\sum_1^M p_m(l_m) - T)$. The monotonicity property ensures that this function has a single minimum over the space of λ. Honeycomb locates the minimum of $f(L, \lambda)$ by iterating over λ using a well-known bracketing technique, golden-section search, for minimization in one-dimension. However, the running time of such a numerical iteration technique cannot be bounded analytically.

Honeycomb achieves a fast, bounded running time through two improvements. First, note that minimizing $f(L, \lambda)$ for a specific value λ' can be done by independently minimizing $C_m(l_m) - \lambda' P_m(l_m)$ for each object. Thus, each iteration can be performed in $O(M \log N)$ time. Second, for each object, the minimum changes at the most $\lceil \log N \rceil$ times while iterating over λ. This implies that we need to minimize $f(L, \lambda)$ for at the most $M \lceil \log N \rceil$ discrete values of λ. These discrete values can be precomputed and sorted at the beginning of the optimization. A binary search over the ordered, discrete space of λ can find the upper and lower bounds in $O(logM)$ time. This yields a numerical algorithm whose overall running time is $O(M \log M \log N)$, including precomputation, sorting, and iterations.

While the optimization algorithm provides an efficient solution technique, it relies on tradeoff information about all objects in the system to compute a globally optimal solution. It is clearly impractical to make information about every object in the system available to every node. At the same time, computing replication levels based solely on locally cached object leads to large deviations from the global optimum. Honeycomb resolves this issue through distributed aggregation mechanisms.

Honeycomb aggregates global tradeoff characteristics by combining objects with similar overhead performance tradeoffs into larger units called *clusters*. Clusters are formed by comparing the ratios of the dominant factors in cost and performance functions. In Cobweb, objects with comparable values for $\frac{q_m}{s_m u_m}$ ratios are clustered and treated as a single unit. CobWeb maintains a constant B number of clusters for each level of replication, that is, all objects replicated at each level are divided into B clusters. These clusters are then aggregated system-wide by exchanging aggregate tradeoff factors for each cluster between the neighbors in the overlay network. Overall, each node determines a close estimate of the global optimal by utilizing the precise overhead and performance information for the locally cached objects, and cluster-level coarse-grained information for other objects.

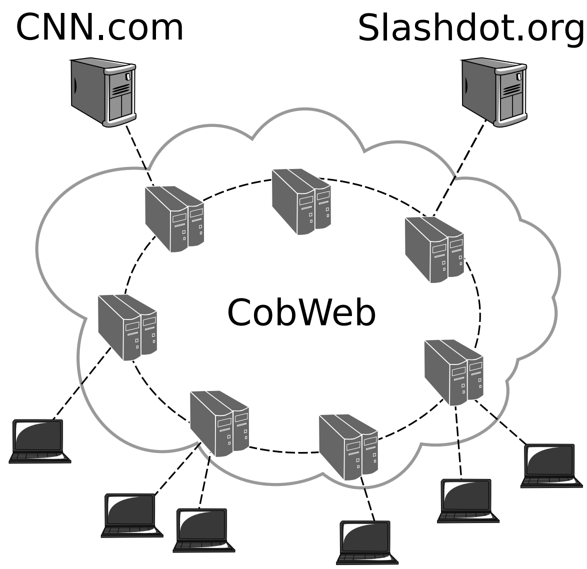

Figure 2: **System Organization**

3 System Architecture

CobWeb operates as a globally distributed ring of cooperating nodes. Each CobWeb node acts as a Web proxy capable of serving any HTTP request. We envision that institutions that currently have large Web caches at their gateway to the Internet, will let the caches join the global CobWeb ring and share cache content intelligently and optimally. Other publicly available Web caches, such as Squid, may also be part of the CobWeb system taking the benefits independent users. The overall architecture of CobWeb is illustrated in Figure 2.

CobWeb distributes objects uniformly between its nodes through consistent-hashing [17]. Each web object is assigned a unique identifier that is a SHA-1 hash of its URL. When a CobWeb proxy receives a request from a client, it routes the request through the underlying overlay, directing the query toward the object's home node, the node whose identifier is numerically closest to the object's identifier. The first node along the routing path which has a copy of the object returns the object to the origin CobWeb proxy, which is responsible for delivering it to the client.

Web objects are not loaded into CobWeb unless requested. When a URL is first requested, its home node is responsible for fetching the object from the origin web server and inserting it into the system. Subsequently, the home node is also responsible for renewing the object when it expires and propagating changes to other nodes. Non-cacheable web objects are simply delivered to the client but not stored within the CobWeb system. Home nodes also delete objects from the system if they do not receive any queries over a long period of time.

CobWeb inherits high failure resilience from the overlay substrate. When a home node fails, the next closest node in the identifier automatically becomes the home node of an object. Objects for which home nodes has the sole copy, simply disappear from the system. This behavior is perfectly correct because CobWeb serves merely as a performance enhancing soft cache, rather than a permanent store. Moreover, popular objects would not be lost in this manner because they will be widely replicated in the neighborhood of the home node.

Users access CobWeb in a transparent way without requiring any extensions or reconfigurations to the browser. In order take advantage of CobWeb, a user merely needs to append ".cobweb.org:8888" to the main URL of a web site. The http request is diverted to the closest CobWeb server through DNS-redirection. Subsequently, all web pages accessed through links on the main URL are automatically redirected through CobWeb. The latter is achieved through URL rewriting. Alternatively, CobWeb is also available as a conventional proxy service, which can be accessed by setting the proxy options in the browser to point to the closest CobWeb node.

An important issue in any cooperative web cache is that a single compromised node can introduce misleading content into the system and launch phishing attacks. While, this is not a problem if CobWeb were to be deployed under centralized management, such as inside Akamai or on Planet-Lab, a collaborative environment poses security risks that need to be tackled. The security issue is further heightened because web objects are not self-certifying. To reduce this vulnerability, we propose a collaborative approach for certifying web content. A small quorum of CobWeb nodes can independently fetch objects and sign objects using a shared key exchanged through threshold cryptographic protocols [34, 15]. Such a collaborative approach prevents individual rogue nodes from introducing corrupt content into the system by mistake or for malice.

The rest of this section describes how resource optimal replication is managed in CobWeb.

3.1 Popularity Aggregation

The optimization algorithm described in Section 2.2 takes as input the performance and cost characteristics of the object. The object-specific cost information, such as the size, update rate, and server imposed load limit, can be stored and replicated along with the object. The workload-specific characteristics, that is, the query rate of an object, on the other hand, needs to be aggregated in the system, since queries are spread over all nodes caching that object.

A naive way to compute the query rate of an object, is to have each node periodically measure, in some *aggregation interval*, the number of queries an object receives in a given period. However, if the query distribution is heavy-tailed, as if often the case in web traffic [3], there can be several orders of magnitude of difference between the query rates of the popular and unpopular objects. Hence, no single aggregation interval would be large enough to accurately estimate the query rates of all objects and small enough to allow the system to detect rapid changes in the popularity of objects, which may arise during a flash-crowd.

One alternative is to measure the inter-arrival times of each object at each node and use those measurements to determine the query rate. However, since objects may be replicated at different nodes, any single node cannot estimate the global query inter-arrival time of an object.

CobWeb uses a hybrid of the two approaches, namely query-rate estimation and inter-arrival time estimation. Nodes with cached objects measure the number of queries for those objects in each aggregation interval. Each node periodically transmits the data collected for each object towards the home node of that object. Each node along the path of the route aggregates the data they receive and continues to route the data toward the home node. Ultimately, each node receives a count of queries for all the objects for which it is the home node. To reduce aggregation overhead, CobWeb sends aggregation messages only if they are non-zero. This reduces the number of aggregation messages sent at each aggregation interval.

Home nodes, then estimate the inter-arrival time using the aggregate query-rate received by it. For unpopular objects which may not be queried for in many aggregation intervals, the home node estimates the query inter-arrival time in terms of the number of aggregation intervals before a query is seen. That is, if an object receives one query every i^{th} aggregation interval, it has a query inter-arrival time of i. For popular objects, which many queries in the same aggregation intervals, it estimates their query inter-arrival time as $1/j$, where j is the number of queries seen in a single aggregation interval. This technique allows us to choose very small values for the aggregation interval, which in turn enables CobWeb to quickly detect changes in the query rate and adapt accordingly.

3.2 Replication Protocol

Given the replica placement solution provided by the numerical solver, CobWeb needs to ensure that objects are replicated at the appropriate levels. To accomplish this, nodes periodically exchange information about objects and their replication levels.

For every object m, a node A is the parent of a node B if A is the next node along the path from B to the home node of m. Periodically, each node sends information to its parents about the replicas that they are caching from their parent nodes. If the parent node determines that a child is lacking an object that it ought to be caching, the parent node sends that object to the child. The parent node also sends the child node a list of objects that the child should continue to cache. Upon receiving this list, the child determines which objects it should no longer cache, and deletes them.

To reduce the cost for this exchange of information, information exchange between parent and child nodes are encoded in bloom filters. These bloom filters are then piggy-backed on aggregation messages that are sent every aggregation interval.

Once a node receives a replica from its parent, it is responsible for independently determining whether the extent of replication for that object needs to be increased. If that is the case, that node will inform its child nodes that they too need to maintain a copy of the replica.

This simple replication protocol allows CobWeb to efficiently handle churn in the network. When a new node joins the system, it obtains all the objects it needs to cache by contacting its parent nodes. When a node leaves the system, the overlay network ensures that its role is automatically taken over by another node.

3.3 Update Propagation

A common concern in maintaining replicas at multiple locations is the issue of maintaining consistency. Due to the structure of its overlay network, CobWeb is capable of efficiently maintaining consistency among objects. When a web object expires, its home node is responsible for fetching a new copy from the origin web server. This new copy is then propagated proactively to all nodes with cached copies of the object. Given the replication level of an object, each node can determine exactly the set of nodes it needs to deliver the updates to, allowing this process to be fast and efficient.

A version number is attached to each object. As objects are refreshed, their version numbers are increased. This allows nodes that miss a proactive update to restore themselves to a consistent state. When parent nodes exchange aggregation messages with child nodes, the version numbers of each object is also passed from the parent node to the child node. This exchange of version number is sent in a compressed message in the following way. The version number and the object identifier of each object are passed through a hash function to create a version key for each object. Version keys are then encoded in a bloom-filter which is sent along with the aggregation message. If a parent node notices that there is a discrepancy between its version key and that of its child, it sends a copy of the object to the child. Upon receiving a copy of an object that has a larger version key, the child node replaces its copy of the object with the new one.

4 Implementation

The previous sections outlined the core distributed algorithms and mechanisms that enable a CDN to achieve high performance while respecting resource consumption constraints. In this section, we describe the CobWeb implementation and show how the algorithmic advantages of the analytical framework can be made practical, transparent and easy to use.

CobWeb is implemented on FreePastry v1.3, an open-source implementation of Pastry [25]. Layering CobWeb on Pastry enables the system to build on the strong failure-resilience, scalability, worst-case performance guarantees provided by Pastry, and to complement these properties with strong average-case performance guarantees.

The CobWeb replication framework is practical and straight-forward to implement. Table 1 shows the size of the different components of the system. The total complexity of the numerical solver, combined with the high-performance web cache front-end, is roughly comparable to the complexity of the Pastry overlay. In fact, most of the complexity resides in mundane issues like HTTP parsing, streaming content from multiple sources to clients, and coordination of concurrent threads, as opposed to the numerical solver.

We envision that CobWeb will be deployed on server-class hosts deployed close to the network core, under a single administrative authority. This is identical to the Akamai model as well as the current deployment model where our research group runs the open CobWeb cache on PlanetLab. Even though CobWeb is built on a peer-to-peer proxy that can integrate any host anywhere, admitting poorly provisioned hosts located behind cable lines into the system is unlikely to offset the additional overhead they entail. Further, in a collaborative deployment, where nodes under different administrative domains are part of the CobWeb network, some nodes may be malicious and either attack the overlay or corrupt the content cached in the system. This problem can be easily solved if web servers provide digitally signed certificates along with content. An alternative solution that does not require changes to servers is to use threshold-cryptography to generate a certificate for content [34, 15]. When new content is to be inserted into the ring, the object can be fetched and partially-signed by a quorum of ring members. If the quorum size exceeds a threshold, partial signatures may be combined into a single signature that attests that t out of n nodes in a wedge on the CobWeb ring agree on the content. Such a scheme can ensure that rogue nodes below a threshold level cannot corrupt the system with bad content and other measures [4] can protect the underlying substrate from malicious nodes. However, the design and implementation of such a threshold-cryptographic scheme for a non-collaborative environment is beyond the scope of this paper.

In the rest of this section, we describe the choices we made in the implementation of CobWeb.

Component	Lines of code
FreePastry	17,712
Numerical Solver	6,163
Web Cache and Proxy	7,798

Table 1: Code complexity of the components of CobWeb

4.1 User Interface

CobWeb provides two different interfaces for different classes of uses. Users that can change the proxy settings in their browser can simply designate a CobWeb node as a proxy. In designating the proxy node, users can either specify the explicit address of a CobWeb node close to them, or instead use the generic proxy address "cobweb.closestnode.com". As described in Section 4.3 below, CobWeb uses the Meridian mechanism [32] based on active measurements to locate the CobWeb node closest to the client.

Although the proxy interface is fast and relatively easy to use, it is not always possible for users to change the proxy settings of their browsers. Further, content providers, such as Slashdot, who wish to take advantage of the load-shedding and performance improvement provided by the CobWeb cache may not be in a position to force their clients to modify their proxy designations. In these cases, clients can be redirected to use the CobWeb cache by appending the suffix "cob-web.org:8888" to the host name of any URL. For instance, the CNN.com can be accessed via the URL "http://www.cnn.com.cob-web.org:8888". Rewriting the host name suffix forces client browsers to look up the name with the CobWeb DNS server, which again uses the Meridian mechanism to route the client's request to the closest CobWeb node.

4.2 URL Rewriting

CobWeb performs URL rewriting on the fly in order to provide clients with a seamless experience, where all resources on a "cobwebbed" URL are fetched from the CobWeb cache instead of the origin server. This enables CobWeb to support high-volume sites such as Slashdot. Consider a web page that consists of a HTML page that is hosted on one server, and many images that are hosted on another server with a different host name. URL-rewriting ensures that when the page is requested through CobWeb, all the images will be accessed through CobWeb as well, alleviating the load on both the HTML server and the image server. The Coral CDN, which does not perform URL rewriting, cannot cache resources that are referenced within HTML pages as absolute URLs. Because the host name is explicitly specified in these URLs, Coral clients will request these resources directly from the origin server instead of through the CDN. Naturally, URL rewriting incurs additional overhead, which we compensate by setting the target performance slightly lower than what it otherwise would be. Note that URL rewriting occurs only once when a page is first fetched by a CobWeb node from the origin server. Subsequent accesses incur no overhead since the resultant page is then cached in the system.

4.3 Proximity Detection

Lookup latency performance is critical in any web cache. To provide low latency performance, it is important that users are directed to the CobWeb proxy that is closest to them. CobWeb accomplishes this by using the Meridian algorithm for closest node selection [32]. When a user first queries for a cobwebbed URL, a DNS request is sent to CobWeb's DNS server, which initiates a recursive Meridian lookup. Under the Meridian scheme, each node maintains a list of peers that are organized into concentric, non-overlapping rings with exponentially increasing radii, based on the node's distance from each of these peers. Each Meridian node determines its distance d to the client using a reverse DNS query or an ICMP ping, examines its rings in the range $d/2$ to $3d/2$ to find suitable peers, and asks those peers to measure their distances to the client. If a suitable peer is found, the query is forwarded to that peer and the process continues recursively; otherwise, the current node is designated as the closest proxy for that client. Note that the Meridian algorithm reduces the distance between the candidate proxy and the client node exponentially at each hop, has been proven to succeed with very high probability under general models for the Internet latency space, and achieves low error rates in practice.

To mask the latency of proximity detection from the client, CobWeb employs caching at two levels. Internally, CobWeb caches internal measurements taken during the Meridian routing process that are used to determine internode distances. In addition, when the closest node to a client is found, the identity of that node is cached at the DNS server for a relatively long period of 5 minutes, allowing subsequent queries from that client to be satisfied instantly.

5 Evaluation

In this section, we evaluate the performance of CobWeb through extensive simulations and measurements from a real world deployment of our system.

5.1 Simulations

We first compare the performance of CobWeb, in its two different configurations, with Beehive, the state of the art in using proactive replication to achieve low latency performance. In addition, we compare CobWeb with PCPastry, a passive caching system, to show the difference in the characteristics of a proactive replication system such as CobWeb and that of a passive, opportunistic caching system. Finally, we examine the performance of CobWeb in the face of "flash crowds" and show that it is capable of quickly adapting to rapid changes in the popularity of objects.

In the experiments below, we run CobWeb in two different configurations. In the first configuration, *CobWeb-TL*, CobWeb is configured to achieve a Target Latency to guarantee high performance. In our experiments, we set this target latency to 0.5 hops, which implies that more than 50% of queries will be satisfied at the local CobWeb proxy. In the second configuration, *CobWeb-TB*, CobWeb is set to meet a target bandwidth limit. This emulates the situation where a CDN needs to provide optimal performance subject to a resource constraint. In our experiment, we set the bandwidth limit to 0.25 kb/s. In both cases, CobWeb-TL and CobWeb-TB are configured with an aggregation interval of 10 minutes.

We compare these two CobWeb configurations to Beehive, a state of the art DHT that provides constant time lookup performance through proactive replication. Unlike CobWeb, however, Beehive does not make use of object sizes and update rates when computing its replica placement, and assumes that queries follow a Zipf distribution. Beehive is also configured to meet the same latency target of 0.5 hops.

Where appropriate, we also compare CobWeb to PC-Pastry, which is a version of Pastry extended to perform common opportunistic caching at each node; every node simply caches all content that it receives. The efficacy of such a passive caching scheme depends entirely on the workload, and typically does not perform well under heavy-tailed popularity distributions. As a baseline for comparison, we also include plain Pastry in our simulations, with no caching at all.

For each of these systems, our simulations model a 1024 node network. We inject queries to these servers based on a workload extracted from the UCB Home IP traces [11]. The workload consists of a total of 409,600 queries for 10,000 objects. The workload distribution follows a Zipf distribution with parameter 0.78. The queries are uniformly divided among the clients, which send queries into the system at a steady rate. The total query rate seen by the system is about 6 queries per second.

5.1.1 Proactive Replication

Figure 3 shows the latency average latency of CobWeb and Beehive systems over the duration of the experiment. As expected, CobWeb-TL and Beehive both converge to the target latency within the first few hours. CobWeb-TB, on the other hand, experiences a slower improvement in latency because it has to stay within its bandwidth limit. CobWeb-TB's performance stabilizes after about 5 hours, at a steady average lookup latency of about 0.8, because the aggressive bandwidth constraints that were placed on CobWeb-TB do not allow it to maintain a sufficient number of replicas to match CobWeb-TL and Beehive's performance.

Figure 4 shows that analytically informed caching can achieve high performance while keeping bandwidth consumption modest. Not surprisingly, CobWeb-TB converges to its target bandwidth limit of 0.25 kb/s very quickly, and its bandwidth consumption remains at this level in the steady state. Both Beehive and CobWeb-TL, which target lookup performance instead of bandwidth consumption, meet their targets with a bandwidth consumption of 0.5 kb/s.

Unlike CobWeb-TB, Beehive and CobWeb-TL experience an initial bandwidth spike. This is a result of the aggressive replication that occurs at the beginning of the experiment, as both systems try to rapidly improve their hit rates to meet their performance goals. Between the two, CobWeb-TL consumes much lower network bandwidth as it converges to its performance target. The reason for this lower overhead is two-fold. First, CobWeb does not require an accurate estimate of the Zipf parameter of the workload, in fact, it does not even assume a Zipf distribution, allowing it to converge to an optimal solution much faster than Beehive. Second, because CobWeb-TL takes object sizes into account when computing its replication solution, it is able to minimize network usage.

Figure 5 shows the storage overhead of each node during the experiment. We observe that the storage overhead of the systems corresponds closely to that of the network overhead. Note that Beehive's storage overhead initially overshoots its steady state value before gradually settling on its steady-state value. This is a result of an overestimation of the optimal amount of replication as Beehive tries to obtain an accurate estimate of the Zipf parameter of the workload distribution. In comparison, CobWeb-TL, by preferentially replicating smaller objects, incurs only about half the overhead of Beehive while achieving the same performance.

The storage overhead of CobWeb-TB is lower because it is indirectly limited by its bandwidth constraint. Although CobWeb-TB's resource consumption limit is defined in terms of network overhead, this creates an indirect limit on storage consumption due to the fact that each replicated object consumes network bandwidth for update propogation and aggregation overhead. When the system reaches a state where all available network bandwidth is being consumed by maintenance overhead in this fashion, CobWeb-TB can no longer cache additional objects.

5.1.2 Comparison to Passive Caching

To observe the difference in the characteristics of a proactive caching system such as CobWeb and that of a passive caching system, we also compare CobWeb-PL to PCPastry and Pastry. Note that for this experiment, we increased the target latency of CobWeb-TL to 1.0. This allows CobWeb-TL to match PCPastry's performance so that we can make a reasonable comparisons of the two systems' resource consumption.

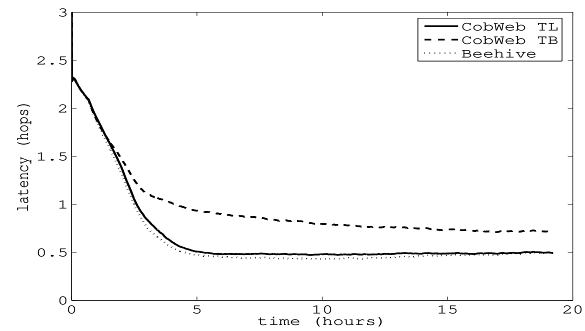

Figure 3: **Average Lookup Latency: Beehive and CobWeb-TL quickly converge to their target latency of 0.5; CobWeb-TB achieves lower performance.**

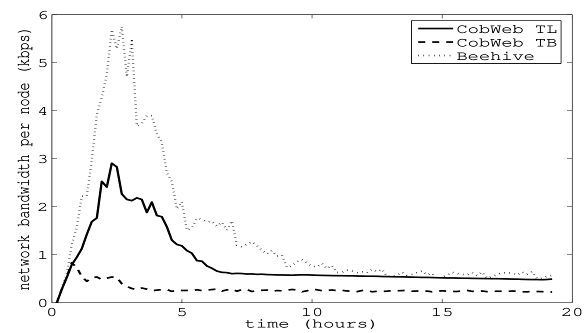

Figure 4: **Per Node Network Overhead: CobWeb-TL incurs significantly lower network overhead than Beehive, while CobWeb-TB uses the least network overhead, being able to stay below its allotted bandwidth limit**

Figure 6 shows the latency performance of PCPastry and CobWeb-TL. We observe that the latency performance of PCPastry converges very slowly to a lookup latency of about 1 hop, as cached copies of objects are slowly created throughout the network in response to the workload. CobWeb-TL converges rapidly to the targeted performance level. Being a passive system, PCPastry is incapable of providing any means of trading off more resources for performance gains.

Although PCPastry and CobWeb-TL provide similar steady state performance, CobWeb-TL is able to accomplish the performance target at much lower cost. Figure 7 shows the storage overhead of the two systems. The storage overhead of PCPastry increases steadily over the course of the experiment. As more queries are injected into the system, the passive caching mechanism of PC-

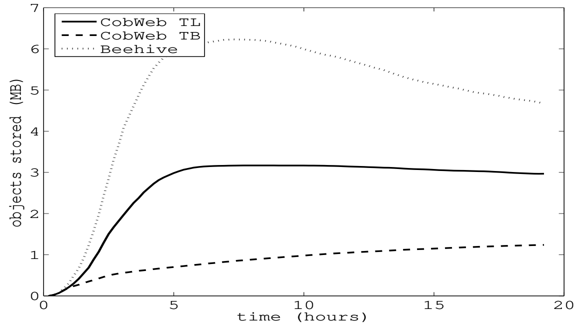

Figure 5: **Per Node Storage Overhead: CobWeb-TL incurs significantly lower storage overhead than Beehive, while CobWeb-TB, because of its bandwidth limit constraint, incurs the least storage overhead.**

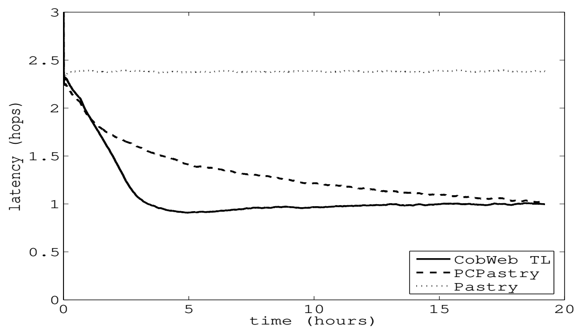

Figure 6: **CobWeb-TL converges to the target latency of 1 rapidly, while PCPastry converges much more slowly.**

Pastry indiscriminately caches every object that passes through every node. In sharp contrast, CobWeb-TL computes an optimal replication strategy and stores a much smaller set of objects at each node. Once CobWeb-TL achieves a steady state where it is able to meet its performance target, its storage overhead remains constant.

5.1.3 Flash Crowds

One of the goals of the CobWeb system is to alleviate the "Slashdot effect", also known as "flash crowds." We simulate the conditions of a flash crowd and show that CobWeb adapts rapidly to such situations. In this experiment, the workload consists of 409,600 queries for a total of 5000 unique objects. The query distribution follows a Zipf distribution with exponent 0.9 and the aggregation interval for CobWeb is set to 45 seconds. The two systems, CobWeb-TL and CobWeb-TB, are configured with a target latency of 1 hop, and a target bandwidth limit of 2 kbps respectively. In order to simulate a flash crowd, the popularities of the 10 least popular objects in the system are increased by three orders of magnitude, after 10 hours, making them the most popular objects in the system.

Figure 8 shows the average latency observed by clients over the course of the experiment. At the 10th hour, when

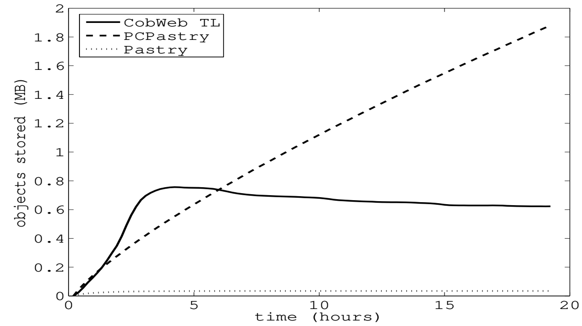

Figure 7: **Per Node Storage Overhead: CobWeb-TL's storage overhead reaches a low, steady-state value rapidly, while PCPastry's storage overhead increases steadily overtime.**

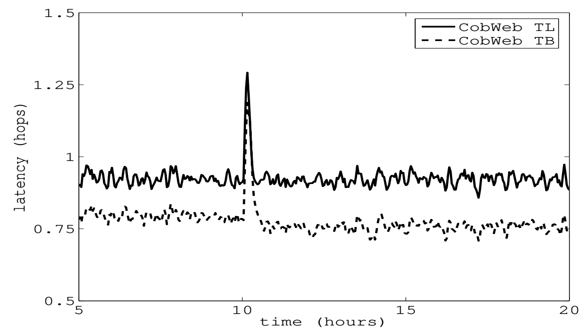

Figure 8: **Network bandwidth consumed during a "flash crowd": CobWeb-TL sees a sudden increase in network bandwidth usage which rapidly returns to its previous steady state; CobWeb-TB shows little change in network bandwidth usage.**

the "flash crowd" occurs, both CobWeb-TL and CobWeb-TB experience a sudden increase in the average latency. However, both systems quickly recover to their steady state average latency within a matter of minutes.

The corresponding network bandwidth consumption is shown in Figure 9. When the "flash crowd" occurs, CobWeb-TL's network bandwidth consumption increases rapidly, because CobWeb-TL aggressively replicates the newly popular objects in order to meet its performance targets. CobWeb-TB, on the other hand, sees only a small increase in its network bandwidth consumption.

Our results show that CobWeb performs well under "flash crowd" conditions. CobWeb's fast aggregation techniques allowed the system to detect changes to object popularity quickly and change replication strategy accordingly. As a result, both CobWeb-TL and CobWeb-TB were able to recover to their steady state performance within minutes. CobWeb-TB was able to accomplish this while staying within its target bandwidth limit.

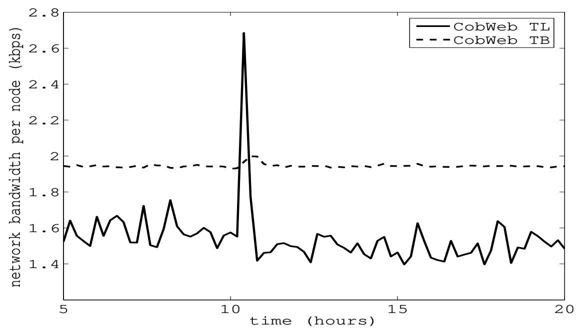

Figure 9: **Average lookup latency during a "flash crowd": Both systems see a small increase in latency, but quickly recovers to the steady state latency.**

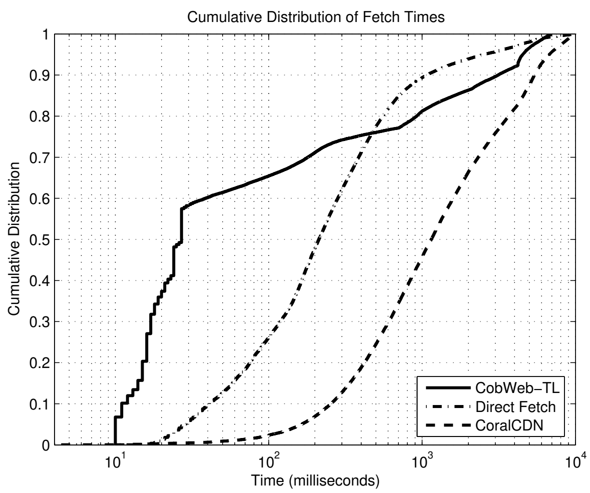

Figure 10: **Cdf of latency to fetch web objects: clients using CobWeb observed a large performance increase over clients fetching web objects directly from web servers.**

5.2 Physical Deployment

We next show results from a live deployment of CobWeb on PlanetLab to demonstrate that the performance benefits seen in simulations are achievable in practice.

Our deployment consists of a set of 90 widely distributed Planet-Lab [2] nodes, each acting as a CobWeb server. Each CobWeb server is configured in CobWeb-TL mode, minimizing network overhead while aiming a target lookup latency of 0.5 hops. We use a workload derived from a week-long trace from a busy proxy server that is part of the IRCache project at the National Laboratory for Applied Network Research [13]. Our trace contains a total of 100,000 queries for 24,713 unique objects. The query distribution closely follows a Zipf-like distribution with parameter 0.83.

We divide this workload uniformly and issue HTTP requests from 20 PlanetLab nodes. The aggregate rate of queries sent into the system is about 240 queries per sec-

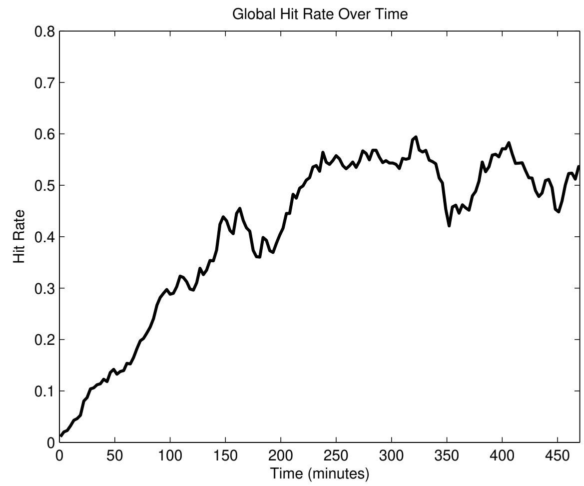

Figure 11: **Hit Rates over time: CobWeb-TL converges to a target hit rate of 0.5**

ond. We measure the time taken to complete each query as seen by each of these clients, as well as the network overhead and cache hit rates seen by each CobWeb server. Next, we repeat the experiment on Coral, another content distribution network deployed on Planet-Lab [10]. Finally, we measure the latency seen by each of the clients when they fetched web objects directly from the origin servers without the use of any web proxies.

We note that CoralCDN is not a performance-oriented content distribution network. Instead, it is designed to alleviate load on popular but poorly provisioned web sites. Therefore, a direct performance comparison between CobWeb and CoralCDN is unfair. Our inclusion of CoralCDN in our measurements serves merely as a indication of the performance of a representative content distribution network deployed on the same platform as CobWeb.

Our experimental results show that CobWeb provides a significant performance improvement over fetching objects directly from the origin server. Figure 10 shows the cumulative distribution of lookup latencies for fetching objects through CobWeb and directly from the origin server. Note that the horizontal axis of the graph is plotted on a log scale. We observe that the cumulative distribution graph for CobWeb rises steeply to about 0.58. This steep rise corresponds to the large portion of queries that were satisfied by a hit in the local cache. Approximately 60% of queries were satisfied in less than 30 milliseconds. In contrast, less than 5% of direct fetches were completed in that time. The graph shows that the median time to fetch an object through CobWeb was 27 milliseconds, while the median time to fetch an object directly from the origin web server was 200 milliseconds.

We also measured the hit rates observed at each of

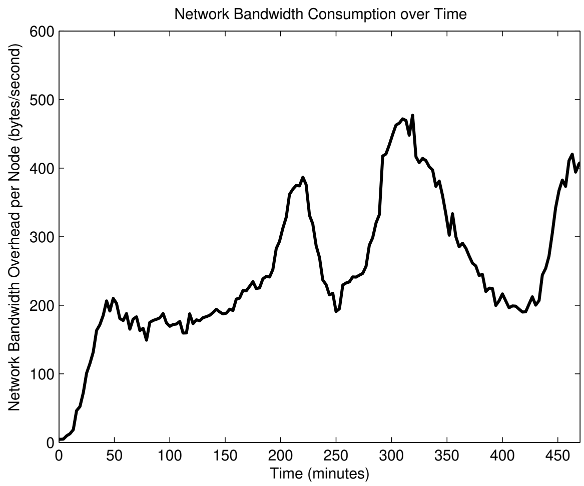

Figure 12: **Network Overhead Per Node: CobWeb incurred a modest network overhead.**

the CobWeb proxies. Figure 11 shows the change in hit rates as the experiment progressed. As CobWeb proxies learned about the object and query distribution, objects begin to be replicated in the network, causing hit rates to gradually increase. We see that after 4 hours, CobWeb's hit rate stabilizes around 0.5. This shows that CobWeb successfully meets its performance targets.

Our measurements shows that the network overhead incurred was modest, never exceeding 500 bytes per second (Figure 12).

CobWeb demonstrates that informed, proactive replication is capable of supporting a high-performance content distribution network that minimizes resource overhead by taking into account object popularity, sizes, and update rate when computing the optimal replication solution. The modest network overhead incurred suggests that CobWeb can scale to support a large population of clients with a high query rate.

5.3 Summary

Overall, our results show that CobWeb performed well under a variety of conditions. Our simulation results show that CobWeb was able to meet performance targets while minimizing resource consumption overhead. In particular, CobWeb-TL successfully achieved a target latency of 0.5, matching that of Beehive while incurring only half the storage overhead. In addition, CobWeb-TB was able to optimize lookup latency while respecting the consumption constraints placed on it.

When compared to passive caching, our results show that CobWeb's proactive replication approach is not only capable of achieving better performance, when set to match PCPastry's steady state performance, CobWeb achieves converges to the desired performance faster, and at a lower overhead cost.

Our simulations also showed that CobWeb was capable of alleviating the decreased performance that is typically observed during a "flash crowd". Using a short aggregation interval, CobWeb was able to react quickly to sharp changes in object popularity and adjust its replication strategy accordingly to meet performance targets. In addition, CobWeb-TB was able to adapt quickly to the "flash crowd" while staying under its bandwidth limit.

Finally, we verified our simulation results through measurements from a real world deployment of CobWeb. Our measurement results show that CobWeb is indeed capable of achieving a high hit rate under a real work load. CobWeb achieved a median fetch latency of 27 milliseconds compared to the median fetch latency of 200 milliseconds for fetching objects directly from the origin server.

6 Related Work

Web caching has been an active area of research, with many proposed algorithms for cooperative caching [6, 30, 16, 24, 9, 20, 28, 21, 10, 29]. However, previous work has mostly been based on opportunistic, passive caching algorithms or heuristic approaches. The CobWeb approach, based on an analysis-driven approach for proactive replication, qualitatively differs from demand-driven caching by proactively propagating replicas, and from existing content distribution networks by relying on a near-optimal solution instead of heuristics for replication.

Recently, several research groups have looked at applying peer-to-peer techniques to web caching. One such system is Squirrel, a peer-to-peer web cache targeted at replacing central demand-side web caches within a local are network [14]. The Squirrel system is a passive, opportunistic cache that is functionally equivalent to the PCPastry system used for comparison in our simulation experiments.

Backslash, a collaborative web mirroring system, also uses a distributed hash table overlay to cache resources. It aims to aggressively replicate popular resources to alleviate "flash crowds". Stavrou et al. propose using a randomized overlay to achieve similar goals [26]. However, both systems focus solely on mitigating the effects of flash crowds and suffer from high latency, making their performance undesirable under normal conditions.

Besides CobWeb, two other CDNs are currently deployed on Planet-Lab. Codeen is an academic test bed CDN that explores different CDN control algorithms and evaluates the design space of heuristics-based CDN redirection algorithms that balance load, locality, and proximity [29]. Coral is a peer-to-peer content distribution network that relies on a hierarchical structure "distributed sloppy hashtables" to reduce the load seen by web servers, shielding ill-provisioned websites from excess traffic.

Beehive is a DHT that uses optimization techniques similar to CobWeb [22]. Beehive is a DHT based on optimal structured replication that is capable of constant time lookup performance. However, Beehive is ill-suited for use in a CDN setting because it makes several critical assumptions about the object and query distribution of the application it serves. First, Beehive assumes that the popularity of objects follows a strict Zipf distribution. Second, Beehive assumes objects are of uniform sizes and thus the replication cost of one object is the same as any other. Third, Beehive does not take into account the update rates of objects, and hence fails to account for the cost of keeping replicated objects consistent. In CobWeb, we have shown how these shortcomings can be overcome by a more general analytical model and a more powerful distributed solver.

7 Conclusion

In this paper, we presented CobWeb, a globally distributed content distribution network that takes a principled approach to the problem of web object caching.

Overall, this paper makes three contributions. First, we formally capture the fundamental tradeoff between performance and cost of web caches in an analytical model, and pose it as a mathematical optimization problem. We propose a novel algorithm and show that the optimization problem can be resolved in a near-optimal fashion.

Second, we show how our analytical model and its numerical solution can be implemented in a distributed fashion on a peer-to-peer substrate. The resulting content distribution network, CobWeb, benefits from the resilience and self-organizing properties of distributed hashtables, allowing it to scale and recover from failures. In addition, CobWeb is able to achieve a target lookup latency while minimizing network and storage overhead, optimize lookup latency while meeting a resource consumption budget, and adapt quickly to changes in workloads.

Finally, through extensive simulations driven by real world trace data, and measurements from a real deployment of the CobWeb system on Planet-Lab, we show that CobWeb is indeed capable of meeting its target performance goals.

References

[1] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing Reference Locality in the WWW. In *Proc. of IEEE International Conference in Parallel and Distributed Information Systems*, Tokyo, Japan, June 1996.

[2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of Symposium on Networked Systems Design and Implementation*, Boston, MA, Mar. 2004.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE International Conference on Computer Communications*, New York, NY, Mar. 1999.

[4] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proc. of Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[5] M. Castro, P. Druschel, C. Hu, and A. Rowstron. Proximity Neighbor Selection in Tree-Based Structured Peer-to-Peer Overlays. Technical Report MSR-TR-2003-52, Microsoft Research, Sept. 2003.

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proc. of USENIX Annual Technical Conference*, pages 153–164, San Diego, CA, Jan. 1996.

[7] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-Based Traces. Technical Report TR-95-010, Boston University, 1995.

[8] F. Douglis, A. Feldman, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997.

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[10] M. Freedman, E. Freudenthal, and D. Mazières. Democratizing Content Publication with Coral. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Mar. 2004.

[11] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proc. of Usenix Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997.

[12] N. Harvey, M. Jons, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar. 2003.

[13] IRCache. http://www.ircache.net.

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A Decentralized Peer-to-Peer Web Cache. In *Proc. ACM Symposium on Principles of Distributed Computing*, Monterey, CA, July 2002.

[15] W. K. Josephson, E. G. Sirer, and F. B. Schneider. Peer-to-Peer Authentication With a Distributed Single Sign-On Service. In *Proc. of International Workshop on Peer-to-Peer Systems*, San Diego, CA, Feb. 2004.

[16] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM Symposium on Theory of Computing*, El Paso, TX, May 1997.

[17] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM Symposium on Theory of Computing*, El Paso, TX, Apr. 1997.

[18] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of ACM Symposium on Principles of Distributed Computing*, Monterey, CA, Aug. 2002.

[19] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. of International Workshop on Peer-to-Peer Systems*, Cambridge, CA, Mar. 2002.

[20] J.-M. Menaud, V. Issarny, and M. Bantre. A Scalable and Efficient Cooperative System for Web Caches. *IEEE Concurrency*, 8(3):56–62, 2000.

[21] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture. In *Proc. of International WWW Caching Workshop*, Manchester, UK, June 1997.

[22] V. Ramasubramanian and E. G. Sirer. Beehive: Exploiting Power Law Query Distributions for 0(1) Lookup Performance in Peer-to-Peer Overlays. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Mar. 2004.

[23] S. Ratnasamy, P. Francis, M. Hadley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[24] K. W. Ross. Hash-Routing for Collections of Shared Web Caches. *IEEE Network Magazine*, 1997.

[25] A. Rowstorn and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001.

[26] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. In *Proc. of IEEE International Conference on Network Protocols*, Paris, France, Nov. 2002.

[27] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[28] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. of International Conference on Distributed Computing Systems*, Austin, TX, May 1999.

[29] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proc. of Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[30] D. Wessels and kc Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, 1998.

[31] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proc. of ACM Symposium on Operating Systems Principles*, Kiawah Island, CA, Dec. 1999.

[32] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: a Lightweight Network Location Service without Virtual Coordinates. In *Proc. of ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[33] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[34] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.