Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays

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

Abstract

Structured peer-to-peer hash tables provide decentralization, self-organization, failure-resilience, and good worst-case lookup performance for applications, but suffer from high latencies (O($logN$)) in the average case. Such high latencies prohibit them from being used in many relevant, demanding applications such as DNS. In this paper, we present a proactive replication framework that can provide constant lookup performance for common Zipf-like query distributions. This framework is based around a closed-form optimal solution that achieves O(1) lookup performance with low storage requirements, bandwidth overhead and network load. Simulations show that this replication framework can realistically achieve good latencies, outperform passive caching, and adapt efficiently to sudden changes in object popularity, also known as flash crowds. This framework provides a feasible substrate for high-performance, low-latency applications, such as peer-to-peer domain name service.

1 Introduction

Peer-to-peer distributed hash tables (DHTs) have recently emerged as a building-block for distributed applications. *Unstructured* DHTs, such as Freenet and the Gnutella network [1, 5], offer decentralization and simplicity of system construction, but may take up to O(N) hops to perform lookups in networks of N nodes. *Structured* DHTs, such as Chord, Pastry, Tapestry and others [14, 17, 18, 21, 22, 24, 28], are particularly well-suited for large scale distributed applications because they are self-organizing, resilient against denial-of-service attacks, and can provide O(log N) lookup performance. However, for large-scale, high-performance, latency-sensitive applications, such as the domain name service (DNS) and the world wide web, this logarithmic performance bound translates into high latencies. Previous work on serving DNS using a peer-to-peer lookup service concluded that, despite their desirable properties, structured DHTs are unsuitable for latency-sensitive applications due to their high lookup costs [8].

In this paper, we describe how proactive replication can be used to achieve constant lookup performance efficiently on top of a standard O(log N) peer-to-peer distributed hash table for certain commonly-encountered query distributions. It is well-known that the query distributions of several popular applications, including DNS and the web, follow a power law distribution [2, 15]. Such a well-characterized query distribution presents an opportunity to optimize the system according to the expected query stream. The critical insight in this paper is that, for query distributions based on a power law, *proactive* (model-driven) replication can enable a DHT system to achieve a small constant lookup latency on average. In contrast, we show that common techniques for *passive* (demand-driven) replication, such as caching objects along a lookup path, fail to make a significant impact on the average-case behavior of the system.

We outline the design of a replication framework, called Beehive, with the following goals:

- **High Performance**: Enable O(1) average-case lookup performance, effectively decoupling the performance of peer-to-peer DHT systems from the size of the network. Provide O(log N) worst-case lookup performance.

- **High Scalability**: Minimize the background traffic in the network to reduce aggregate network load and per-node bandwidth consumption. Keep the memory and disk space requirements at each peer to a minimum.

- **High Adaptivity**: Adjust the performance of the system to the popularity distribution of objects. Respond quickly when object popularities change, as with flash crowds and the “slashdot effect.”

Beehive achieves these goals through efficient proactive replication. By proactive replication, we mean actively propagating copies of objects among the nodes participating in the network. There is a fundamental tradeoff between replication and resource consumption: more copies of an object will generally improve lookup performance at the cost of space, bandwidth and aggregate network load.

Beehive performs this tradeoff through an informed analytical model. This model provides a closed-form, optimal solution that guarantees O(1), constant-time lookup performance with the minimum number of object replicas. The particular constant C targeted by the system is tunable. Beehive enables the system designer to specify a fractional value. Setting C to a fractional value, such as 0.5, ensures that 50% of queries will be satisfied at the source, without any additional network hops. Consequently, Beehive implements a sub-one hop hash-table. The value of C can be adjusted dynamically to meet real-time performance goals.

Beehive uses the minimal number of replicas required to achieve a targeted performance level. Minimizing replicas reduces storage requirements at the peers, lowers bandwidth consumption and load in the network, and enables cache-coherent updates to be performed efficiently. Beehive uses low-overhead protocols for tracking, propagating and updating replicas. Finally, Beehive leverages the structure of the underlying DHT to update objects efficiently at runtime, and guarantees that subsequent lookups will return the latest copy of the object.

While this paper describes the Beehive proactive replication framework in its general form, we use the domain name system as a driving application. Several shortcomings of the current, hierarchical structure of DNS makes it an ideal application candidate for Beehive. First, DNS is highly latency-sensitive, and poses a significant challenge to serve efficiently. Second, the hierarchical organization of DNS leads to a disproportionate amount of load being placed at the higher levels of the hierarchy. Third, the higher nodes in the DNS hierarchy serve as easy targets for distributed denial-of-service attacks and form a security vulnerability for the entire system. Finally, nameservers required for the internal leaves of the DNS hierarchy incur expensive administrative costs, as they need to be manually administered and secured. Peer-to-peer DHTs address all but the first critical problem; we show in this paper that Beehive's replication strategy can address the first.

We have implemented a prototype Beehive-based DNS server on Pastry [22]. We envision that the DNS nameservers that are currently used to serve small, dedicated portions of the naming hierarchy would form a Beehive network and collectively serve the namespace. While we use DNS as a guiding application and demonstrate that serving DNS with DHT is feasible, we note that a full treatment of the implementation of an alternative peer-to-peer DNS system is beyond the scope of this paper, and focus instead on the general-purpose Beehive framework for proactive replication. The framework is sufficiently general to achieve O(1) lookup performance in other settings, including web caching, where the query distribution follows a power law.

Overall, this paper describes the design of a replication framework that enables constant lookup performance in structured DHTs for common query distributions, applies it to a P2P DNS implementation, and makes the following contributions. First, it proposes proactive replication of objects and provides a closed-form analytical solution for the optimal number of replicas needed to achieve O(1) lookup performance. The storage, bandwidth and load placed on the network by this scheme are modest. In contrast, we show that simple caching strategies based on passive replication incur large ongoing costs. Second, it outlines the design of a complete system based around this analytical model. This system is layered on top of Pastry, an existing peer-to-peer substrate. It includes techniques for estimating the requisite inputs for the analytical model, mechanisms for replica distribution and deletion, and fast update propagation. Finally, it presents results from a prototype implementation of a peer-to-peer DNS service to show that the system achieves good performance, has low overhead, and can adapt quickly to flash crowds. In turn, these approaches enable the benefits of P2P systems, such as self-organization and resilience against denial of service attacks, to be applied to latency-sensitive applications.

The rest of this paper is organized as follows. Section 2 provides a broad overview of our approach and describes the storage and bandwidth-efficient replication components of Beehive in detail. Section 3 describes our implementation of Beehive over Pastry. Section 4 presents the results and expected benefits of using Beehive to serve DNS queries. Section 5 surveys different DHT systems and summarizes other approaches to caching and replication in peer-to-peer systems Section 6 describes future work and Section 7 summarizes our contributions.

2 The Beehive System

Beehive is a general replication framework that operates on top of any DHT that uses prefix-routing [19], such as Chord [24], Pastry [22], Tapestry [28], and Kademlia [18]. Such DHTs operate in the following manner. Each node has a unique randomly assigned identifier in a circular identifier space. Each object also has a

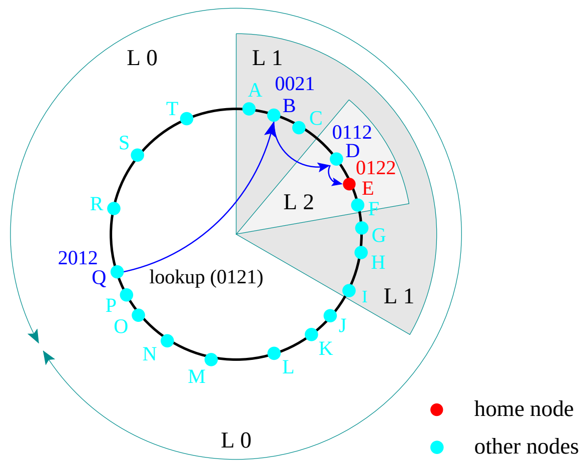

Figure 1: **This figure illustrates the levels of replication in Beehive. A query for object 0121 takes three hops from node Q to node E, the home node of the object. By replicating the object at level 2, that is at D and F, the query latency can be reduced to two hops. In general, an object replicated at level i incurs at most i hops for a lookup.**

unique randomly selected identifier, and is stored at the node whose identifier is closest to its own, called the *home node*. Routing is performed by successively matching a prefix of the object identifier against node identifiers. Generally, each step in routing takes the query to a node that has one more matching prefix than the previous node. A query traveling k hops reaches a node that has k matching prefixes[1]. Since the search space is reduced exponentially, this query routing approach provides O(log_bN) lookup performance on average, where N is the number of nodes in the DHT and b is the base, or fanout, used in the system.

The central observation behind Beehive is that the length of the average query path will be reduced by one hop when an object is proactively replicated at all nodes logically preceding that node on all query paths. For example, replicating the object at all nodes one hop prior to the home-node decreases the lookup latency by one hop. We can apply this iteratively to disseminate objects widely throughout the system. Replicating an object at all nodes k hops or lesser from the home node will reduce the lookup latency by k hops. The Beehive replication mechanism is a general extension of this observation to find the appropriate amount of replication for each object based on its popularity.

Beehive controls the extent of replication in the system by assigning a *replication level* to each object. An object at level i is replicated on all nodes that have at least i matching prefixes with the object. Queries to objects replicated at level i incur a lookup latency of at most i hops. Objects stored only at their home nodes are at level log_bN, while objects replicated at level 0 are cached at all the nodes in the system. Figure 1 illustrates the concept of replication levels.

The goal of Beehive's replication strategy is to find the minimal replication level for each object such that the average lookup performance for the system is a constant C number of hops. Naturally, the optimal strategy involves replicating more popular objects at lower levels (on more nodes) and less popular objects at higher levels. By judiciously choosing the replication level for each object, we can achieve constant lookup time with minimal storage and bandwidth overhead.

Beehive employs several mechanisms and protocols to find and maintain appropriate levels of replication for its objects. First, an analytical model provides Beehive with closed-form optimal solutions indicating the appropriate levels of replication for each object. Second, a monitoring protocol based on local measurements and limited aggregation estimates relative object popularity, and the global properties of the query distribution. These estimates are used, independently and in a distributed fashion, as inputs to the analytical model which yields the locally desired level of replication for each object. Finally, a replication protocol proactively makes copies of the desired objects around the network. The rest of this section describes each of these components in detail.

2.1 Analytical Model

In this section, we provide a model that analyzes Zipf-like query distributions and provides closed-form optimal replication levels for the objects in order to achieve constant average lookup performance with low storage and bandwidth overhead.

In Zipf-like, or power law, query distributions, the number of queries to the i^{th} most popular object is proportional to $i^{-\alpha}$, where α is the parameter of the distribution. The query distribution has a heavier tail for smaller values of the parameter α. A Zipf distribution with parameter 0 corresponds to a uniform distribution. The total number of queries to the most popular m objects, $Q(m)$, is approximately $\frac{m^{1-\alpha}-1}{1-\alpha}$ for $\alpha \neq 1$, and $Q(m) \simeq ln(m)$ for $\alpha = 1$.

Using the above estimate for the number of queries received by objects, we pose an optimization problem to minimize the total number of replicas with the constraint that the average lookup latency is a constant C.

Let b be the base of the underlying DHT system, M the number of objects, and N the number of nodes in the system. Initially, all M objects in the system are stored

[1]Strictly speaking, the nodes encountered towards the end of the query routing process may not share progressively more prefixes with the object, but remain numerically close. This detail does not significantly impact either the time complexity of standard DHTs or our replication algorithm. Section 3 discusses the issue in more detail.

only at their home nodes, that is, they are replicated at level $k = log_b N$. Let x_i denote the fraction of objects replicated at level i or lower. From this definition, x_k is 1, since all objects are replicated at level k. Mx_0 most popular objects are replicated at all the nodes in the system.

Each object replicated at level i is cached in N/b^i nodes. $Mx_i - Mx_{i-1}$ objects are replicated on nodes that have exactly i matching prefixes. Therefore, the average number of objects replicated at each node is given by $Mx_0 + \frac{M(x_1-x_0)}{b} + \cdots + \frac{M(x_k-x_{k-1})}{b^k}$. Consequently, the average per node storage requirement for replication is:

$$M[(1-\frac{1}{b})(x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k-1}}{b^{k-1}}) + \frac{1}{b^k}] \quad (1)$$

The fraction of queries, $Q(Mx_i)$, that arrive for the most popular Mx_i objects is approximately $\frac{(Mx_i)^{1-\alpha}-1}{M^{1-\alpha}-1}$. The number of objects that are replicated at level i is $Mx_i - Mx_{i-1}, 0 < i \leq k$. Therefore, the number of queries that travel i hops is $Q(Mx_i) - Q(Mx_{i-1}), 0 < i \leq k$. The average lookup latency of the entire system can be given by $\sum_{i=1}^{k} i(Q(Mx_i) - Q(Mx_{i-1}))$. The constraint on the average latency is $\sum_{i=1}^{k} i(Q(Mx_i) - Q(Mx_{i-1})) \leq C$, where C is the required constant lookup performance. After substituting the approximation for $Q(m)$, we arrive at the following optimization problem.

$$\text{Minimize } x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k-1}}{b^{k-1}}, \text{ such that} \quad (2)$$
$$x_0^{1-\alpha} + x_1^{1-\alpha} + \cdots + x_{k-1}^{1-\alpha} \geq k - C(1 - \frac{1}{M^{1-\alpha}}) \quad (3)$$
$$\text{and } x_0 \leq x_1 \leq \cdots \leq x_{k-1} \leq 1 \quad (4)$$

Note that constraint 4 effectively reduces to $x_{k-1} \leq 1$, since any optimal solution to the problem with just constraint 3 would satisfy $x_0 \leq x_1 \leq \cdots \leq x_{k-1}$.

We can use the Lagrange multiplier technique to find an analytical closed-form optimal solution to the above problem with just constraint 3, since it defines a convex feasible space. However, the resulting solution may not guarantee the second constraint, $x_{k-1} \leq 1$. If the obtained solution violates the second constraint, we can force x_{k-1} to 1 and apply the Lagrange multiplier technique to the modified problem. We can obtain the optimal solution by repeating this process iteratively until the second constraint is satisfied. However, the symmetric property of the first constraint facilitates an easier analytical approach to solve the optimization problem without iterations.

Assume that in the optimal solution to the problem, $x_0 \leq x_1 \leq \cdots \leq x_{k'-1} < 1$, for some $k' \leq k$, and $x_{k'} = x_{k'+1} = \cdots = x_k = 1$. Then we can restate the optimization problem as follows:

$$\text{Minimize } x_0 + \frac{x_1}{b} + \cdots + \frac{x_{k'-1}}{b^{k'-1}}, \text{ such that} \quad (5)$$
$$x_0^{1-\alpha} + x_1^{1-\alpha} + \cdots + x_{k'-1}^{1-\alpha} \geq k' - C', \quad (6)$$
$$\text{where } C' = C(1 - \frac{1}{M^{1-\alpha}})$$

This leads to the following closed-form solution:

$$x_i^* = [\frac{d^i(k'-C')}{1+d+\cdots+d^{k'-1}}]^{\frac{1}{1-\alpha}}, \forall 0 \leq i < k' \quad (7)$$
$$x_i^* = 1, \forall k' \leq i \leq k \quad (8)$$
$$\text{where } d = b^{\frac{1-\alpha}{\alpha}}$$

We can derive the value of k' by satisfying the condition that $x_{k'-1} < 1$, that is, $\frac{d^{k'-1}(k'-C')}{1+d+\cdots+d^{k'-1}} < 1$.

As an example, consider a DHT with base 32, $\alpha = 0.9$, $10,000$ nodes, and $1,000,000$ objects. Applying this analytical method to achieve an average lookup time, C, of one hop yields $k' = 2$, $x_0 = 0.001102, x_1 = 0.0519$, and $x_2 = 1$. Thus, the most popular 1102 objects would be replicated at level 0, the next most popular 50814 objects would be replicated at level 1, and all the remaining objects at level 2. The average per node storage requirement of this system would be 3700 objects.

The optimal solution obtained by this model applies only to the case $\alpha < 1$. For $\alpha > 1$, the closed-form solution will yield a level of replication that will achieve the target lookup performance, but the amount of replication may not be optimal because the feasible space is no longer convex. For $\alpha = 1$, we can obtain the optimal solution by using the approximation $Q(m) = \ln m$ and applying the same technique. The optimal solution for this case is as follows:

$$x_i^* = \frac{M^{\frac{-C}{k'}} b^i}{b^{\frac{k'-1}{2}}}, \forall 0 \leq i < k' \quad (9)$$
$$x_i^* = 1, \forall k' \leq i \leq k \quad (10)$$

This analytical solution has three properties that are useful for guiding the extent of proactive replication. First, the analytical model provides a solution to achieve any desired constant lookup performance. The system can be tailored, and the amount of overall replication controlled, for any level of performance by adjusting C over a continuous range. Since structured DHTs preferentially keep physically nearby hosts in their top-level routing tables, even a large value for C can dramatically speed up end-to-end query latencies [4]. Second, for a large class of query distributions ($\alpha \leq 1$), the solution provided by this model achieves the optimal number of object replicas required to provide the desired performance. Minimizing the number of replicas reduces per-node storage requirements, bandwidth consumption and aggregate load on the network. Finally, k' serves as an upper bound for the worst case lookup time for any successful query, since all objects are replicated at least in level k'.

We make two assumptions in the analytical model: all objects incur similar costs for replication, and objects do not change very frequently. For applications such as DNS, which have essentially homogeneous object sizes and whose update-driven traffic is a very small fraction of the replication overhead, the analytical model provides an efficient solution. Applying the Beehive approach to applications such as the web, which has a wide range of object sizes and frequent object updates, may require an extension of the model to incorporate size and update frequency information for each object. A simple solution to standardize object sizes is to replicate object pointers instead of the objects themselves. While effective and optimal, this approach adds an extra hop to C and violates sub-one hop routing.

2.2 Popularity and Zipf-Parameter Estimation

The analytical model described in the previous section requires estimates of the α parameter of the query distribution and the relative popularities of the objects. Beehive employs a combination of local measurement and limited aggregation to keep track of these parameters and adapt the replication appropriately.

Beehive nodes locally measure the number of queries received for each object on that node. For non-replicated objects, the count at the home node reflects the popularity of that object. However, queries for an object replicated at level i are evenly distributed across approximately N/b^i nodes. In order to estimate popularity of such an object as accurately as an unreplicated object, one would need an N/b^i-fold increase in the measurement interval. Since dilating the sampling interval would prevent the system from reacting quickly to changes in object popularity, Beehive aggregates popularity data from multiple nodes to arrive at accurate estimates of object popularity within a relatively short sampling interval.

Aggregation in Beehive takes place periodically, once every *aggregation interval*. Each node A sends to node B in the i^{th} level of its routing table an *aggregation message* containing the number of accesses of each object replicated at level i or lower and having $i+1$ matching prefixes with B. When node B receives these messages from all nodes at level i, it aggregates the access counts and sends the result to all nodes in the $(i+1)^{th}$ level of its routing table. This process allows popularity data to flow towards the home node.

A similar process operates in reverse to disseminate the aggregated totals from the higher levels towards the nodes at the leaves. A node at level $i+1$ sends the latest aggregated estimate of access counts to nodes at level i for that object. This exchange occurs when the higher level node is contacted by lower level node for aggregation. For an object replicated at level i, it takes $2(\log N - i)$ rounds of aggregation to complete the information flow from the leaves up to the home node and back.

The overall Zipf-parameter, α, is also estimated in a similar manner. Each node locally estimates α using linear regression to compute the slope of the best fit line, since a Zipf-like popularity distribution is a straight line in log-scale. Beehive nodes then refine this estimate by averaging it with the local estimates of other nodes they communicate with during aggregation.

There will be fluctuations in the estimation of access frequency and the Zipf parameter due to temporal variations in the query distribution. In order to avoid large discontinuous changes to an estimate, we age the estimate using exponential decay.

2.3 Replication Protocol

Beehive requires a protocol to replicate objects at the replication levels determined by the analytical model. In order to be deployable in wide area networks, the replication protocol should be asynchronous and not require expensive mechanisms such as distributed consensus or agreement. In this section, we describe an efficient protocol that enables Beehive to replicate objects across a DHT.

Beehive's replication protocol uses an asynchronous and distributed algorithm to implement the optimal solution provided by the analytical model. Each node is responsible for replicating an object on other nodes at most one hop away from itself; that is, at nodes that share one less prefix than the current node. Initially, each object is replicated only at the home node at a level $k = log_b N$, where N is the number of nodes in the system and b is the base of the DHT, and shares k prefixes with the object. If an object needs to be replicated at the next level $k-1$, the home node pushes the object to all nodes that share one less prefix with the home node. Each of the level $k-1$ nodes at which the object is currently replicated may independently decide to replicate the object further, and push the object to other nodes that share one less prefix with it. Nodes continue the process of independent and distributed replication until all the objects are replicated at appropriate levels. In this algorithm, nodes

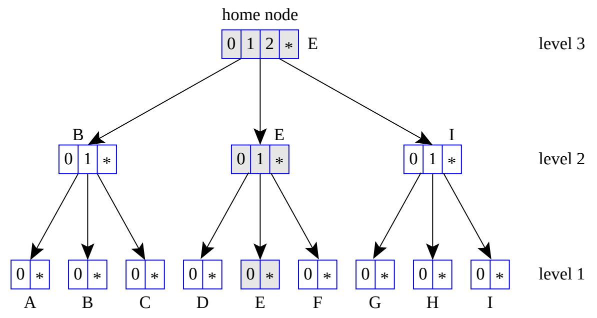

Figure 2: **This figure illustrates how the object 0121 at its home node E is replicated to level 1. For nodes A through I, the numbers indicate the prefixes that match the object identifier at different levels. Each node pushes the object independently to nodes with one less matching digit.**

that share $i+1$ prefixes with an object are responsible for replicating that object at level i, and are called i *level deciding nodes* for that object. For each object replicated at level i at some node A, the i level deciding node is that node in its routing table at level i that has matching $i+1$ prefixes with the object. For some objects, the deciding node may be the node A itself.

This distributed replication algorithm is illustrated in Figure 2. Initially, an object with identifier 0121 is replicated at its home node E at level 3 and shares 3 prefixes with it. If the analytical model indicates that this object should be replicated at level 2, node E pushes the objects to nodes B and I with which it shares 2 prefixes. Node E is the level 2 deciding node for the object at nodes B, E, and I. Based on the popularity of the object, the level 2 nodes B, E, and I may independently decide to replicate the object at level 1. If node B decides to do so, it pushes a copy of the object to nodes A and C with which it shares 1 prefix and becomes the level 1 deciding node for the object at nodes A, B, and C. Similarly, node E may replicate the object at level 1 by pushing a copy to nodes D and F, and node I to G and F.

Our replication algorithm does not require any agreement in the estimation of relative popularity among the nodes. Consequently, some objects may be replicated partially due to small variations in the estimate of the relative popularity. For example in Figure 2, node E might decide not to push object 0121 to level 1. We tolerate this inaccuracy to keep the replication protocol efficient and practical. In the evaluation section, we show that this inaccuracy in the replication protocol does not produce any noticeable difference in performance.

Beehive implements this distributed replication algorithm in two phases, an *analysis phase* and a *replicate phase*, that follow the aggregation phase. During the analysis phase, each node uses the analytical model and the latest known estimate of the Zipf-parameter α to obtain a new solution. Each node then locally changes the replication levels of the objects according to the solution. The solution specifies for each level i, the fraction of objects, x_i that need to be replicated at level i or lower. Hence, $\frac{x_i}{x_{i+1}}$ fraction of objects replicated at level $i+1$ or lower should be replicated at level i or lower. Based on the current popularity, each node sorts all the objects at level $i+1$ or lower for which it is the i level deciding node. It chooses the most popular $\frac{x_i}{x_{i+1}}$ fraction of these objects and locally changes the replication level of the chosen objects to i, if their current replication level is $i+1$. The node also changes the replication level of the objects that are not chosen to $i+1$, if their current replication level is i or lower.

After the analysis phase, the replication level of some objects could increase or decrease, since the popularity of objects changes with time. If the replication level of an object decreases from level $i+1$ to i, it needs to be replicated in nodes that share one less prefix with it. If the replication level of an object increases from level i to $i+1$, the nodes with only i matching prefixes need to delete the replica. The *replicate phase* is responsible for enforcing the correct extent of replication for an object as determined by the analysis phase. During the replicate phase, each node A sends to each node B in the ith level of its routing table, a *replication message* listing the identifiers of all objects for which B is the i level deciding node. When B receives this message from A, it checks the list of identifiers and pushes to node A any unlisted object whose current level of replication is i or lower. In addition, B sends back to A the identifiers of objects no longer replicated at level i. Upon receiving this message, A removes the listed objects.

Beehive nodes invoke the analysis and the replicate phases periodically. The analysis phase is invoked once every *analysis interval* and the replicate phase once every *replication interval*. In order to improve the efficiency of the replication protocol and reduce load on the network, we integrate the replication phase with the aggregation protocol. We perform this integration by setting the same durations for the replication interval and the aggregation interval and combining the replication and the aggregation messages as follows: When node A sends an aggregation message to B, the message contains the list of objects replicated at A whose i level deciding node is B. Similarly, when node B replies to the replication message from A, it adds the aggregated access frequency information for all objects listed in the replication message.

The analysis phase estimates the relative popularity of the objects using the estimates for access frequency obtained through the aggregation protocol. Recall that, for an object replicated at level i, it takes $2(logN - i)$ rounds of aggregation to obtain an accurate estimate of the access frequency. In order to allow time for the information flow during aggregation, we set the replication interval to at least $2logN$ times the aggregation interval.

Random variations in the query distribution will lead to fluctuations in the relative popularity estimates of objects, and may cause frequent changes in the replication levels of objects. This behavior may increase the object transfer activity and impose substantial load on the network. Increasing the duration of the aggregation interval is not an efficient solution because it decreases the responsiveness of system to changes. Beehive limits the impact of fluctuations by employing hysteresis. During the analysis phase, when a node sorts the objects at level i based on their popularity, the access frequencies of objects already replicated at level $i-1$ is increased by a small fraction. This biases the system towards maintaining already existing replicas when the popularity difference between two objects is small.

The replication protocol also enables Beehive to maintain appropriate replication levels for objects when new nodes join and others leave the system. When a new node joins the system, it obtains the replicas of objects it needs to store by initiating a replicate phase of the replication protocol. If the new node already has objects replicated when it was previously part of the system, then these objects need not be fetched again from the deciding nodes. A node leaving the system does not directly affect Beehive. If the leaving node is a deciding node for some objects, the underlying DHT chooses a new deciding node for these objects when it repairs the routing table.

2.4 Mutable Objects

Beehive directly supports mutable objects by proactively disseminating object updates to the replicas in the system. The semantics of read and update operations on objects is an important issue to consider while supporting object mutability. Strong consistency semantics require that once an object is updated, all subsequent queries to that object only return the modified object. Achieving strong consistency is challenging in a distributed system with replicated objects, because each copy of the replicated object should be updated or invalidated upon object modification. In Beehive, we exploit the structure of the underlying DHT to efficiently disseminate object updates to all the nodes carrying replicas of the object. Our scheme guarantees that when an object is modified, all replicas will be consistently updated at all nodes.

Beehive associates a *version number* with each object to identify modified objects. An object replica with higher version number is more recent than a replica with lower version number. The owner of an object in the system can modify the object by inserting a fresh copy of the object with a higher version number at the home node. The home node proactively propagates the update to all the replicas of the objects using the routing table. If the object is replicated at level i, the home node sends a copy of the updated object to each node B in the i^{th} level of the routing table. Node B then propagates the update to each node in the $(i+1)^{th}$ level of its routing table.

The update propagation protocol ensures that each node A sharing at least i prefixes with the object obtain a copy of the modified object. The object update reaches the node A following exactly the same path a query issued at the object's home node for node A's identifier would follow. Because of this property, all nodes with a replica of the object get exactly one copy of the modified object. Hence, this scheme is both efficient and provides guaranteed cache coherency in the absence of nodes leaving the system.

Nodes leaving the system may cause temporary inconsistencies in the routing table. Consequently, updates may not reach some nodes where objects are replicated. Beehive alleviates this problem by incorporating a lazy update propagation mechanism to the replicate phase. Each node includes in the replication message, the current version numbers of the replicated objects. Upon receiving this message, the deciding node pushes a copy of the object if it has a more recent version.

3 Implementation

Beehive is a general replication mechanism that can be applied to any prefix-based distributed hash table. We have layered our implementation on top of Pastry, a freely available DHT with log(N) lookup performance. Our implementation is structured as a transparent layer on top of FreePastry 1.3, supports a traditional insert/modify/delete/query DHT interface for applications, and required no modifications to underlying Pastry. However, converting the preceding discussion into a concrete implementation of the Beehive framework, building a DNS application on top, and combining the framework with Pastry required some practical considerations and identified some optimization opportunities.

Beehive needs to maintain some additional, modest amount of state in order to track the replication level, freshness, and popularity of objects. Each Beehive node stores all replicated objects in an object repository. Beehive associates the following meta-information with each object in the system, and each Beehive node maintains the following fields within each object in its repository:

- Object-ID: A 128-bit field uniquely identifies the object and helps resolve queries. The object identifier is derived from the hash key at the time of insertion, just as in Pastry.

- Version-ID: A 64-bit version number differentiates fresh copies of an object from older copies cached in the network.
- Home-Node: A single bit specifies whether the current node is the home node of the object.
- Replication-Level: A small integer specifies the current, local replication level of the object.
- Access-Frequency: An integer monitors the number of queries that have reached this node. It is incremented by one for each locally observed query, and reset at each aggregation.
- Aggregate-Popularity: A integer used in the aggregation phase to collect and sum up the access count from all dependent nodes for which this node is the home node. We also maintain an older aggregate popularity count for aging.

In addition to the state associated with each object, Beehive nodes also maintain a running estimate of the Zipf parameter. Overall, the storage cost consists of several bytes per object, and the processing cost of keeping the meta-data up to date is small.

Pastry's query routing deviates from the model described earlier in the paper because it is not entirely prefix-based and uniform. Since Pastry maps each object to the numerically closest node in the identifier space, it is possible for an object to not share any prefixes with its home node. For example, in a network with two nodes 298 and 315, Pastry will store an object with identifier 304 on node 298. Since a query for object 304 propagated by prefix matching alone cannot reach the home node, Pastry completes the query with the aid of an auxiliary data structure called *leaf set*. The leaf set is used in the last few hops to directly locate the numerically closest node to the queried object. Pastry initially routes a query using entries in the routing table, and may route the last couple of hops using the leaf set entries. This required us to modify Beehive's replication protocol to replicate objects at the leaf set nodes as follows. Since the leaf set is most likely to be used for the last hop, we replicate objects in the leaf set nodes only at the highest replication levels. Let $k = log_b N$ be the highest replication level for Beehive, that is, the default replication level for an object replicated only at its home node. As part of the replicate phase, a node A sends a replication message to all nodes B in its routing table as well as its leaf set with a list of identifiers of objects replicated at level $k-1$ whose deciding node is B. B is the deciding node of an object homed at node A, if A would forward a query to that object to node B next. Upon receiving a maintenance message at level $k-1$, node B would push an object to node A only if node A and the object have at least $k-1$ matching prefixes. Once an object is replicated on a leaf set node at level $k-1$, further replication to lower levels follow the replication protocol described in Section 2. This slight modification to Beehive enables it to work on top of Pastry. Other routing metrics for DHT substrates, such as the XOR metric [18], have been proposed that do not exhibit this non-uniformity, and where the Beehive implementation would be simpler.

Pastry's implementation provides two opportunities for optimization, which improve Beehive's impact and reduce its overhead. First, Pastry nodes preferentially populate their routing tables with nodes that are in physical proximity [4]. For instance, a node with identifier 100 has the opportunity to pick either of two nodes 200 and 201 when routing based on the first digit. Pastry selects the node with the lowest network latency, as measured by the packet round-trip time. As the prefixes get longer, node density drops and each node has progressively less freedom to find and choose between nearby nodes. This means that a significant fraction of the lookup latency experienced by a Pastry lookup is incurred on the last hop. Hence, selecting even a large number of constant hops, C, as Beehive's performance target, will have a significant effect on the real performance of the system. While we pick $C = 1$ in our implementation, note that C is a continuous variable and may be set to a fractional value, to get average lookup performance that is a fraction of a hop. $C = 0$ yields a solution that will replicate all objects at all hops, which is suitable only if the total hash table size is small.

The second optimization opportunity stems from the periodic maintenance messages used by Beehive and Pastry. Beehive requires periodic communication between nodes and the member of their routing table and leaf-set for replica dissemination and data aggregation. Pastry nodes periodically send heart-beat messages to nodes in their routing table and leaf set to detect node failures. They also perform periodic network latency measurements to nodes in their routing table in order to obtain closer routing table entries. We can improve Beehive's efficiency by combining the periodic heart-beat messages sent by Pastry with the periodic aggregation messages sent by Beehive. By piggy-backing the ith row routing table entries on to the Beehive aggregation message at replication level i, a single message can simultaneously serve as a heart beat message, Pastry maintenance message, and a Beehive aggregation message.

We have built a prototype DNS name server on top of Beehive in order to evaluate the caching strategy proposed in this paper. Beehive-DNS uses the Beehive framework to proactively disseminate DNS resource records containing name to IP address bindings. The Beehive-DNS server currently supports UDP-based queries and is compatible with widely-deployed resolver libraries. Queries that are not satisfied within the Beehive system are looked up in the legacy DNS by the home node and are inserted into the Beehive framework. The Beehive system stores and disseminates resource records to the appropriate replication levels by monitoring the DNS query stream. Clients are free to route their queries through any node that is part of the Beehive-DNS. Since the DNS system relies entirely on aggressive caching in order to scale, it provides very loose coherency semantics, and limits the rate at which updates can be performed. Recall that the Beehive system enables resource records to be modified at any time, and disseminates the new resource records to all caching name servers as part of the update operation. However, for this process to be initiated, name owners would have to directly notify the home node of changes to the name to IP address binding. We expect that, for some time to come, Beehive will be an adjunct system layered on top of legacy DNS, and therefore name owners who are not part of Beehive will not know to contact the system. For this reason, our current implementation delineates between names that exist solely in Beehive versus resource records originally inserted from legacy DNS. In the current implementation, the home node checks for the validity of each legacy DNS entry by issuing a DNS query for the domain when the time-to-live field of that entry is expired. If the DNS mapping has changed, the home node detects the update and propagates it as usual. Note that this strategy preserves DNS semantics and is quite efficient because only the home nodes check the validity of each entry, while replicas retain all mappings unless invalidated.

Overall, the Beehive implementation adds only a modest amount of overhead and complexity to peer-to-peer distributed hash tables. Our prototype implementation of Beehive-DNS is only 3500 lines of code, compared to the 17500 lines of code for Pastry.

4 Evaluation

In this section, we evaluate the performance costs and benefits of the Beehive replication framework. We examine Beehive's performance in the context of a DNS system and show that Beehive can robustly and efficiently achieve its targeted lookup performance. We also show that Beehive can adapt to sudden, drastic changes in the popularity of objects as well as global shifts in the parameter of the query distribution, and continue to provide good lookup performance.

We compare the performance of Beehive with that of pure Pastry and Pastry enhanced by passive caching. By passive caching, we mean caching objects along all nodes on the query path, similar to the scheme proposed in [23]. We impose no restrictions on the size of the cache used in passive caching. We follow the DNS cache model to handle mutable objects, and associate a time to live with each object. Objects are removed from the cache upon expiration of the time to live.

4.1 Setup

We evaluate Beehive using simulations, driven by a DNS survey and trace data. The simulations were performed using the same source code as our implementation. Each simulation run was started by seeding the network with just a single copy of each object, and then querying for objects according to a DNS trace. We compared the proactive replication of Beehive to passive caching in Pastry (PC-Pastry), as well as regular Pastry.

Since passive caching relies on expiration times for coherency, and since both Beehive and Pastry need to perform extra work in the presence of updates, we conducted a large-scale survey to determine the distribution of TTL values for DNS resource records and to compute the rate of change of DNS entries. Our survey spanned July through September 2003, and periodically queried web servers for the resource records of 594059 unique domain names, collected by crawling the Yahoo! and the DMOZ.ORG web directories. We used the distribution of the returned time-to-live values to determine the lifetimes of the resource records in our simulation. We measured the rate of change in DNS entries by repeating the DNS survey periodically, and derived an object lifetime distribution. We used this distribution to introduce a new version of an object at the home node.

We used the DNS trace [15] collected at MIT between 4 and 11 December 2000. This trace spans $4,160,954$ lookups over 7 days featuring 1233 distinct clients and $302,032$ distinct fully-qualified names. In order to reduce the memory consumption of the simulations, we scale the number of distant objects to 40960, and issue queries at the same rate of 7 queries per sec. The rate of issue for requests has little impact on the hit rate achieved by Beehive, which is dominated mostly by the performance of the analytical model, parameter estimation, and rate of updates. The overall query distribution of this trace follows an approximate Zipf-like distribution with parameter 0.91 [15]. We separately evaluate Beehive's robustness in the face of changes in this parameter.

We performed our evaluations by running the Beehive implementation on Pastry in simulator mode with 1024 nodes. For Pastry, we set the base to be 16, the leaf-set size to be 24, and the length of identifiers to be 128, as recommended in [22]. In all our evalua-

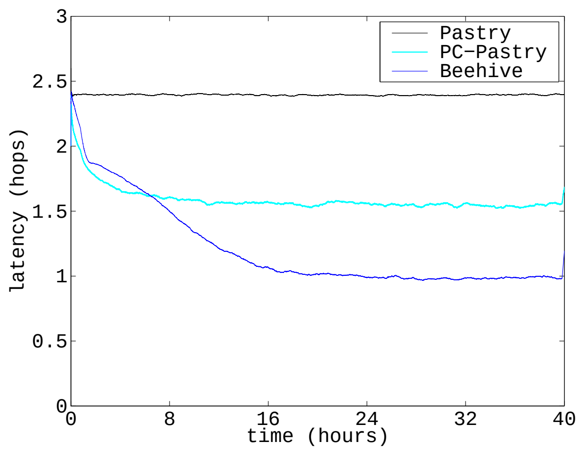

Figure 3: **Latency (hops) vs Time. The average lookup performance of Beehive converges to the targeted $C = 1$ hop after two replication phases.**

tions, the Beehive aggregation and replication intervals were 48 minutes and the analysis interval was 480 minutes. The replication phases at each node were randomly staggered to approximate the behavior of independent, non-synchronized hosts. We set the target lookup performance of Beehive to average 1 hop.

Beehive Performance

Figure 3 shows the average lookup latency for Pastry, PC-Pastry, and Beehive over a query period spanning 40 hours. We plot the lookup latency as a moving average over 48 minutes. The average lookup latency of pure Pastry is about 2.34 hops. The average lookup latency of PC-Pastry drops steeply during the first 4 hours and averages 1.54 after 40 hours. The average lookup performance of Beehive decreases steadily and converges to about 0.98 hops, within 5% of the target lookup performance. Beehive achieves the target performance in about 16 hours and 48 minutes, the time required for two analysis phases followed by a replication phase at each node. These three phases, combined, enable Beehive to propagate the popular objects to their respective replication levels and achieve the expected payoff. In contrast, PC-Pastry provides limited benefits, despite an infinite-sized cache. There are two reasons for the relative ineffectiveness of passive caching. First, the heavy tail in Zipf-like distributions implies that there will be many objects for which there will be few requests, where queries will take many disjoint paths in the network until they collide on a node on which the object has been cached. Second, PC-Pastry relies on time-to-live values for cache coherency, instead of tracking the location of cached objects. The time-to-live values are set conservatively in order to reflect the worst case scenario under which the record may

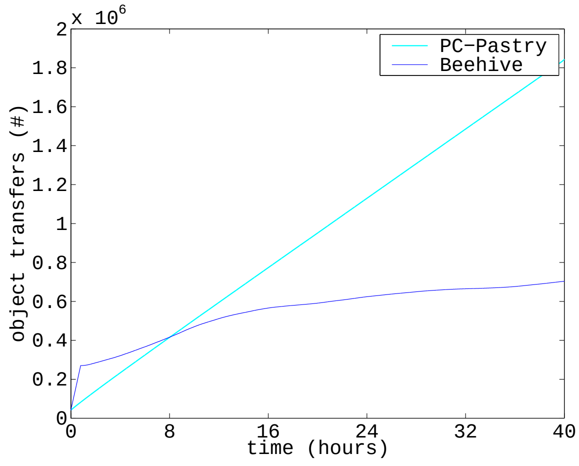

Figure 4: **Object Transfers (cumulative) vs Time. The total amount of object transfers imposed by Beehive is significantly lower compared to caching. Passive caching incurs large costs in order to check freshness of entries in the presence of conservative timeouts.**

be updated, as opposed to the expected lifetime of the object. Our survey indicates that 95% of the DNS records have a lifetime of less than one day, whereas fewer than 0.8% of the records change in 24 hours. Consequently, passive caching suffers from a low hit rate as entries are evicted due to conservative values of TTL set by name owners.

Next, we examine the bandwidth consumed and the network load incurred by PC-Pastry and Beehive, and show that Beehive generates significantly lower background traffic due to object transfers compared to passive caching. Figure 4 shows the total amount of objects transferred by Beehive and PC-Pastry since the beginning of the experiment. PC-Pastry has an average object transfer rate proportional to its lookup latency, since it transfers an object to each node along the query path. Beehive incurs a high rate of object transfer during the initial period; but once Beehive achieves its target lookup performance, it incurs considerably lower overhead, as it needs to perform transfers only in response to changes in object popularity and, relatively infrequently for DNS, to object updates. Beehive continues to perform limited amounts of object replication, due to fluctuations in the popularity of the objects as well as estimation errors not dampened down by hysteresis. The rate of object transfers is initially high because the entire system is started at the same time with only one copy of each object. In practice, node-joins and object-inserts would be staggered in time allowing the system to operate smoothly without sudden spikes in bandwidth consumption.

The average number of objects stored at each node at the end of 40 hours is 380 for Beehive and 420 for pas-

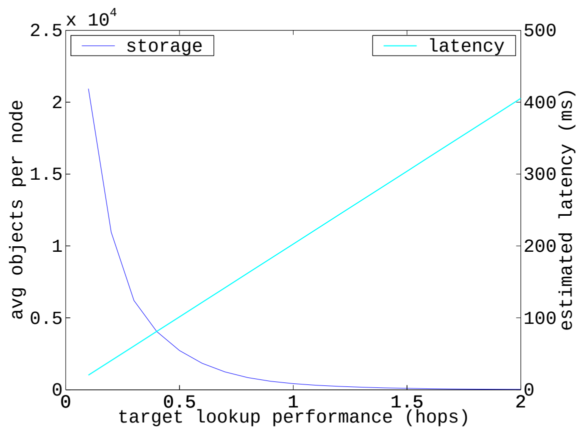

Figure 5: **Storage Requirement vs Latency. This graph shows the average per node storage required by Beehive and the estimated latency for different target lookup performance. This graph captures the trade off between the overhead incurred by Beehive and the lookup performance achieved.**

sive caching. PC-Pastry caches more objects than Beehive even though its lookup performance is worse, due to the heavy tailed nature of Zipf distributions. Beehive requires only 95 objects per node to provide 1.54 hops, the lookup performance achieved by PC-Pastry. Our evaluation shows that Beehive provides 1 hop average lookup latency with low storage and bandwidth overhead.

Beehive efficiently trades off storage and bandwidth for improved lookup latency. Our replication framework enables administrators to tune this trade off by varying the target lookup performance of the system. Figure 5 shows the trade off between storage requirement and estimated latency for different target lookup performance. We used the analytical model described in Section 2 to estimate the storage requirements. We estimated the expected lookup latency from round trip time obtained by pinging all pairs of nodes in PlanetLab, and adding to this 0.42 ms for accessing the local DNS resolver. The average 1 hop round trip time between nodes in PlanetLab is 202.2 ms (median 81.9 ms). In our large scale DNS survey, the average DNS lookup latency was 255.9 ms (median 112 ms). Beehive with a target performance of 1 hop can provide better lookup latency than DNS.

Flash Crowds

Next, we examine the performance of proactive and passive caching in response to changes in object popularity. We modify the trace to suddenly reverse the popularities of all the objects in the system. That is, the least popular object becomes the most popular object, the second least popular object becomes the second most popular object, and so on. This represents a worst case scenario

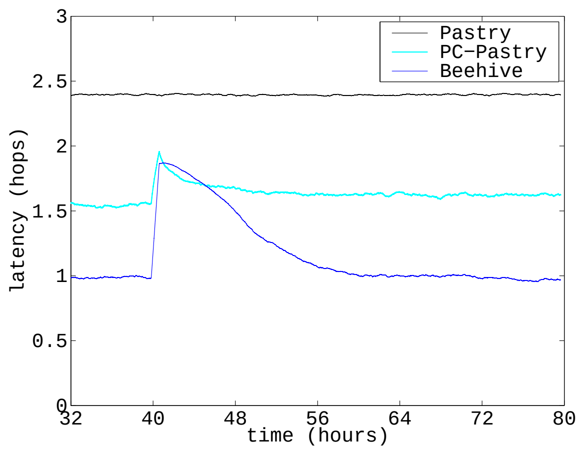

Figure 6: **Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the popularity of objects and brings the average lookup performance to one hop.**

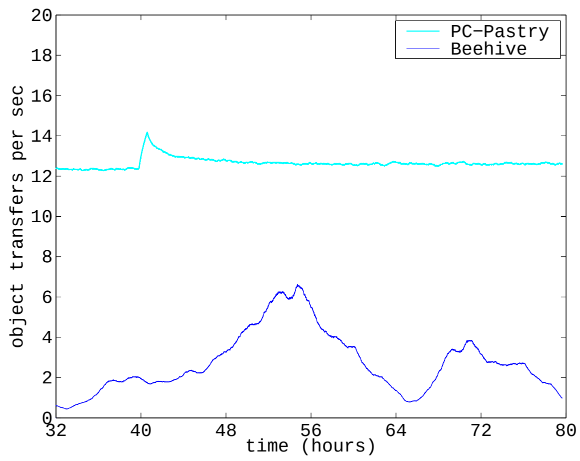

Figure 7: **Rate of Object Transfers vs Time. This graph shows that when popularity of the objects change, Beehive imposes extra bandwidth overhead temporarily to replicate the newly popular objects and maintain constant lookup time.**

for proactive replication, as objects that are least replicated suddenly need to be replicated widely, and vice versa. The switch occurs at $t = 40$, and we issue queries from the reversed popularity distribution for another 40 hours.

Figure 6 shows the lookup performance of Pastry, PC-Pastry and Beehive in response to flash crowds. Popularity reversal causes a temporary increase in average latency for both Beehive and PC-Pastry. Beehive adjusts the replication levels of its objects appropriately and reduces the average lookup performance to about 1 hop after two replication intervals. The lookup performance of passive caching also decreases to about 1.6 hops. Fig-

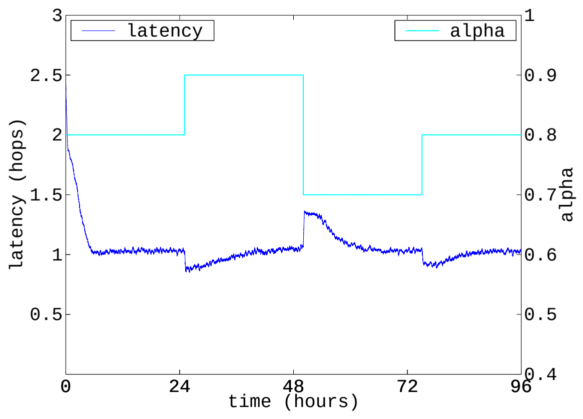

Figure 8: **Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the parameter of the query distribution and brings the average lookup performance to one hop.**

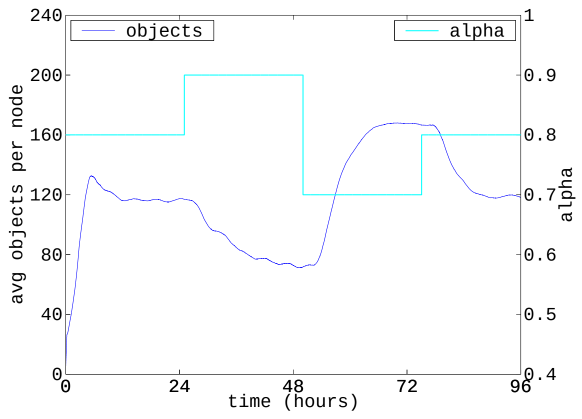

Figure 9: **Objects stored per node vs Time. This graph shows that when the parameter of the query distribution changes, Beehive adjusts the number of replicated objects to maintain O(1) lookup performance with storage efficiency.**

ure 7 shows the instantaneous rate of object transfer induced by the popularity reversal for Beehive and PC-Pastry. The popularity reversal causes a temporary increase in the object transfer activity of Beehive as it adjusts the replication levels of the objects appropriately. Even though Beehive incurs this high rate of activity in response to a worst-case scenario, it consumes less bandwidth and imposes less aggregate load compared to passive caching.

Zipf Parameter Change

Finally, we examine the adaptation of Beehive to global changes in the parameter of the overall query distribution. We issue queries from Zipf-like distributions generated with different values of the parameter α at each 24 hour interval. We seeded these simulations with 4096 objects.

Figure 8 shows the lookup performance of Beehive as it adapts to changes in the parameter of the query distribution. Beehive effectively detects changes in $alpha$ and redistributes object replicas to meet targeted performance objectives. Figure 9 shows the average number of objects replicated at each node in the system by Beehive. As α varies, so do the number of replicas Beehive creates on each node. Beehive increases the number of replicated objects per node as α decreases to meet the targeted performance goal, and reduces the number of replicas as $alpha$ increases to reclaim space. The number of replicas per node shown in Figure 9 agrees with optimal solution provided by the analytical model. Overall, continuously monitoring and estimating the α of the query distribution enables Beehive to adjust the extent and level of replication to compensate for drastic, global changes.

Summary

In this section, we have evaluated the performance of the Beehive replication framework for different scenarios in the context of DNS. Our evaluation indicates that Beehive achieves O(1) lookup performance with low storage and bandwidth overhead. In particular, it outperforms passive caching in terms of average latency, storage requirements, network load and bandwidth consumption. Beehive continuously monitors the popularity of the objects and the parameter of the query distribution, and quickly adapts its performance to changing conditions.

5 Related Work

Unstructured peer-to-peer systems, such as Freenet [5] and Gnutella [1] locate objects through on generic graph traversal algorithms, such as iterative depth-first search and flooding-based breadth-first search, respectively. These algorithms are inefficient, do not scale well, and do not provide sublinear bounds on lookup performance.

Several structured peer-to-peer systems, which provide a worst-case bound on lookup performance, have been proposed recently. CAN [21] maps both objects and nodes on a d-dimensional torus and provides O($dn^{\frac{1}{d}}$) lookup performance. Plaxton et al. [19] introduce a randomized lookup algorithm based on prefix matching to locate objects in a distributed network in O($logN$) probabilistic time. Chord [24], Pastry [22], and Tapestry [28] use consistent hashing to map objects to nodes and use Plaxton's prefix-matching algorithms to achieve O($logN$) worst-case lookup performance. Kademlia [24] also provides O($logN$) lookup performance using a similar search technique, but uses the XOR metric for routing. Viceroy [17] provides O($logN$) lookup performance with a constant degree routing graph. De Bruijn graphs [16, 26] provide O($logN$) lookup performance with 2 neighbors per node and O($logN/loglogN$) with $logN$ degree per node. Beehive can be applied to any of the overlays based on prefix-matching.

A few recently introduced DHTs provide O(1) lookup performance. Kelips [12] probabilistically provides O(1) lookup performance by dividing the network into O($\sqrt{N}$) affinity groups of O($\sqrt{N}$) nodes, replicating every object on every node within an affinity group, and using gossip to propagate updates. An alternative method [13] relies on maintaining full routing state, that is, a complete list of all system members, at each node. Farsite [10] uses routing tables of size O($dn^{\frac{1}{d}}$) to route in O(d) hops, but does not address rapid membership changes. Beehive fundamentally differs from these systems in three fundamental ways. First, Beehive can serve queries in less than one hop on average. And second, it achieves low storage overhead, bandwidth consumption and network load by minimizing the amount of replicated data in the system. Finally, Beehive provides a fine grain control of the trade off between lookup performance and overhead by allowing users to choose the target lookup performance from a continuous range.

Some DHTs pick routing table entries based on network proximity. Recent work [4, 27] has shown that this method can reduce the total lookup latency to a constant. However, the performance improvement is limited in large networks because the achieved absolute latency is several times more than the average single-hop latency.

Several peer-to-peer applications, such as PAST [23] and CFS [9], incorporate caching and replication. Both reserve a part of the storage space at each node to cache query results on the lookup path in order to improve subsequent queries. They also maintain a constant number of replicas of each object in the system in order to improve fault tolerance. As shown in this paper, passive caching schemes achieve limited improvement and do not provide provide performance guarantees.

Some systems employ a combination of caching with proactive object updates. In [6], the authors describe a proactive cache for DNS records where the cache proactively refreshes DNS records as they expire. While this technique reduces the impact of short expiration times on lookup performance, it introduces a large amount of overhead and does not qualitatively improve lookup performance. Controlled Update Propagation (CUP) [20] is a demand-based caching mechanism with proactive object updates. CUP nodes propagate object updates away from a designated home node in accordance to a popularity based *incentive* that flows from the leaf nodes towards the home node. While there are some similarities between the replication protocols of CUP and Beehive, the decision to cache objects and propagate updates in CUP are based on heuristics. [25] describes a distributed hierarchical web cache that replicates objects proactively, selects replica locations based on heuristics and pushes updates to replicas.

The closest work to Beehive is [7], which examines optimal strategies for replicating objects in unstructured peer-to-peer systems. This paper analytically derives the optimal number of randomly-placed object replicas in unstructured peer-to-peer systems. The observations in this work are not directly applicable to structured DHTs, because it assumes that the lookup time for an object depends only on the number of replicas and not the placement strategy. Beehive achieves higher performance with fewer replicas by exploiting the structure of the underlying overlay.

6 Future Work

This paper has investigated the potential performance benefits of model-driven proactive caching and has shown that it is feasible to use peer-to-peer systems in cooperative low-latency, high-performance environments. Deploying full-blown applications, such as a complete peer-to-peer DNS replacement, on top of this substrate will require substantial further effort. Most notably, security issues need to be addressed before peer-to-peer systems can be deployed widely. At the application level, this involves using some authentication technique, such as DNSSEC [11], to securely delegate name service to nodes in a peer to peer system. At the underlying DHT layer, secure routing techniques [3] are required to limit the impact of malicious nodes on the DHT. Both of these techniques will add additional latencies, which may be offset at the cost of additional bandwidth, storage and load by setting Beehive's target performance to lower values. At the Beehive layer, the proactive replication layer needs to be protected from nodes that misreport the popularity of objects. Since a malicious peer in Beehive can replicate an object, or indirectly cause an object to be replicated, at b nodes that have that malicious node in their routing tables, we expect that one can limit the amount of damage that attackers can cause through misreported object popularities.

7 Conclusion

Structured DHTs offer many desirable properties for a large class of applications, including self-organization, failure resilience, high scalability, and a worst-case performance bound. However, their O($logN$) hop average-case performance has prohibited them from serving latency-sensitive applications.

In this paper, we outline a framework for proactive replication that offers O(1) DHT lookup performance for a frequently encountered class of query distributions. At the core of this framework is an analytical model that yields the optimal object replication required to achieve constant time lookups. Beehive achieves high performance by decoupling lookup performance from the size of the network. It adapts quickly to flash crowds, detects changes in the global query distribution and self-adjusts to retain its performance guarantees. Overall, Beehive enables DHTs to be used for serving latency-sensitive applications.

Acknowledgments

We are grateful to Robert Morris, Hari Balakrishnan, Jaeyon Jung, and Emil Sit for providing us with their DNS traces.

References

[1] "The Gnutella Protocol Specification v.0.4." *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*, Mar 2001.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications." *IEEE INFOCOM 1999*, New York NY, Mar 1999.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. "Secure Routing for Structured Peer-to-Peer Overlay Networks." *OSDI 2002*, Boston MA, Dec 2002.

[4] M. Castro, P. Druschel, C. Hu, and A. Rowstron. "Exploiting Network Proximity in Peer-to-Peer Overlay Networks." *Technical Report MSR-TR-2002-82, Microsoft Research*, May 2002.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." *Lecture Notes in Computer Science*, 2009:46-66, 2001.

[6] E. Cohen and H. Kaplan. "Proactive Caching of DNS Records: Addressing a Performance Bottleneck." *SAINT 2001*, San Diego CA, Jan 2001.

[7] E. Cohen and S. Shenker. "Replication Strategies in Unstructured Peer-to-Peer Networks." *ACM SIGCOMM 2002*, Pittsburgh PA, Aug 2002.

[8] R. Cox, A. Muthitacharoen, and R. Morris. "Serving DNS using a Peer-to-Peer Lookup Service." *IPTPS 2002*, Cambridge MA, Mar 2002.

[9] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. "Wide-Area Cooperative Storage with CFS." *ACM SOSP 2001*, Banff Alberta, Canada, Oct 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. "Reclaiming Space from Duplicate Files in a Serverless Distributed File System." *ICDCS 2002*, Vienna, Austria, Jul 2002.

[11] D. Eastlake. "Domain Name System Security Extensions". *Request for Comments (RFC) 2535*, 3^{rd} edition, Mar 1999.

[12] I. Gupta, K. Birman, P. Linga, A. Demers, and R. v. Rennesse. "Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead." *IPTPS 2003*, Berkeley CA, Feb 2003.

[13] A. Gupta, B. Liskov, R. Rodrigues. "One Hop Lookups for Peer-to-Peer Overlays." *HotOS 2003*. Lihue HI, May 2003.

[14] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. "SkipNet: A Scalable Overlay Network with Practical Locality Properties.", *USITS 2003*, Seattle WA, Mar 2003.

[15] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. "DNS Performance and Effectiveness of Caching." *ACM SIGCOMM Internet Measurement Workshop 2001*, San Francisco CA, Nov 2001.

[16] F. Kaashoek and D. Karger. "Koorde: A Simple Degree-Optimal Distributed Hash Table." *IPTPS 2003*, Berkeley CA, Feb 2003.

[17] D. Malkhi, M. Naor, and D. Ratajczak. "Viceroy: A Scalable and Dynamic Emulation of the Butterfly." *ACM PODC 2002*, Monterey CA, Aug 2002.

[18] P. Maymounkov and D. Maziéres. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric." *IPTPS 2002*, Cambridge MA, Mar 2002.

[19] G. Plaxton, R. Rajaraman, and A. Richa. "Accessing nearby copies of replicated objects in a distributed environment." *Theory of Computing Systems*, 32:241-280, 1999.

[20] M. Roussopoulos and M. Baker. "CUP: Controlled Update Propagation in Peer-to-Peer Networks." *USENIX 2003 Annual Technical Conference*, San Antonio TX, Jun 2003.

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

[22] A. Rowstorn and P. Druschel. "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems." *IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov 2001.

[23] A. Rowstorn and P. Druschel. "Storage Management and Caching in PAST, a Large-Scale Persistent Peer-to-Peer Storage Utility." *ACM SOSP 2001*, Banff Alberta, Canada, Oct 2001.

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

[25] R. Tewari, M. Dahlin, H. Vin, and J. Kay. "Design Considerations for Distributed Caching on the Internet." *ICDCS 1999*, Austin TX, Jun 1999.

[26] U. Wieder and M. Naor. "A Simple Fault Tolerant Distributed Hash Table." *IPTPS 2003*, Berkeley CA, Feb 2003.

[27] H. Zhang, A. Goel, and R. Govindan. "Incrementally Improving Lookup Latency in Distributed Hash Table Systems." *SIGMETRICS 2003*, San Diego CA, Jun 2003.

[28] 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, JSAC*, 2003.