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

Venugopalan Ramasubramanian and Emin Gün Sirer

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 achieve O(1) lookup performance for common Zipf-like query distributions. This framework is based around a closed-form 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 [5, 1], 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 [24, 22, 26, 21, 18, 17, 14], are particularly well-suited for large scale distributed applications because they are self-organizing, resilient against denial-of-service attacks, and provide O(log N) lookup performance in both the worst- and the average case. 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 high average-case lookup costs render current structured DHTs unsuitable for latency-sensitive applications, such as DNS [8].

In this paper, we describe how proactive replication can be used to achieve O(1) 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 [15, 2]. 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 three 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. Ensure that the amount of memory and/or disk space required of each peer in the network is kept to a minimum.
- **High Adaptivity**: Promptly adjust the performance of the system in response to changes in the aggregate popularity distribution of objects. Further, cheaply track and maintain the popularity of individual objects in the system to quickly respond when a certain object becomes highly popular, 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. In the limit, proactively copying *all* objects in the DHT to *all* nodes would enable every query to be satisfied in constant time. However, this would not scale to large systems since it would require prohibitive amounts of space on each node, the network would be overloaded during replica creation, and changes to mutable objects would require O(N) updates. In contrast, Beehive performs this tradeoff through an analytical model that provides a closed-form, optimal solution that achieves O(1) lookup performance for power law query distributions while minimizing the number of object copies, and hence reducing storage, bandwidth and load, in the network.

Beehive relies on cheap, local measurements and efficient lease-based protocols for replica coordination. Each node in Beehive continually performs local measurements to determine the relative popularity of the objects in the system, as well as

to estimate global properties of the aggregate query distribution function. Beehive nodes decide how many replicas of each object should be propagated by combining the closed-form solutions from the analytical model with their measurements of the aggregate query distribution function and estimates of object rank. This estimation is performed independently and periodically at each node, while a replica management protocol efficiently propagates or removes cached objects without excessive messaging, global synchronization or agreement.

Objects in Beehive may be modified dynamically. In general, mutable objects pose cache-coherency problems for any replication technique, as older, out-of-date copies of an object may remain cached throughout a system and keep clients from accessing more recent versions. To provide up to date views in the presence of updates, a system needs to track all replicas of an object and either invalidate old copies or propagate the changes when the object is modified. In Beehive, the structured nature of the underlying DHT allows the system to keep track of the placement of all replicas with a single integer. This enables Beehive to efficiently find and update all replicas when an object is modified. Consequently, objects may be updated at any time in Beehive, and lookups performed after an update has completed 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 target application, perform our evaluation with DNS data, and demonstrate that serving DNS lookups with a peer-to-peer distributed hash table is feasible. 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, secure and constantly online. 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 layered on top of the Pastry peer-to-peer hash table [22]. Our prototype implementation is compatible with current client resolver libraries deployed around the Internet. 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 manage the entire namespace. Our implementation supports the existing naming scheme by falling back on legacy DNS when Beehive-DNS lookups fail. Unlike legacy DNS, which relies on cache timeouts for loose coherency and incurs ongoing cache expiration and refill overheads, Beehive-DNS enables resource records to be updated at any time. While we use DNS as a guiding application for evaluating our system, 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 aggregate query distribution follows a power law, similar to DNS.

Overall, this paper describes the design of a replication framework that enables O(1) 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 number of replicas needed to achieve constant-time lookup performance with low costs. 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 propagation and deletion, and a strategy for mapping between the continuous solution in the analytical model and the discrete implementation in Pastry. 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, such as DNS.

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 can be applied to structured DHTs based on prefix-routing [19], such as Chord, Pastry, Tapestry, and Kademlia. These DHTs operate in the following manner. Each node has a unique randomly assigned identifier in a circular identifier space. Each object also has a unique randomly selected identifier assigned from the same space and is stored at the closest node, called the *home node*. Routing is performed by successively matching a prefix of the object identifier against node identifiers. Generally, each step in the query processing 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

[1] Strictly speaking, the nodes encountered towards the end of the query routing process in a sparsely populated DHT may not share progressively more pre-

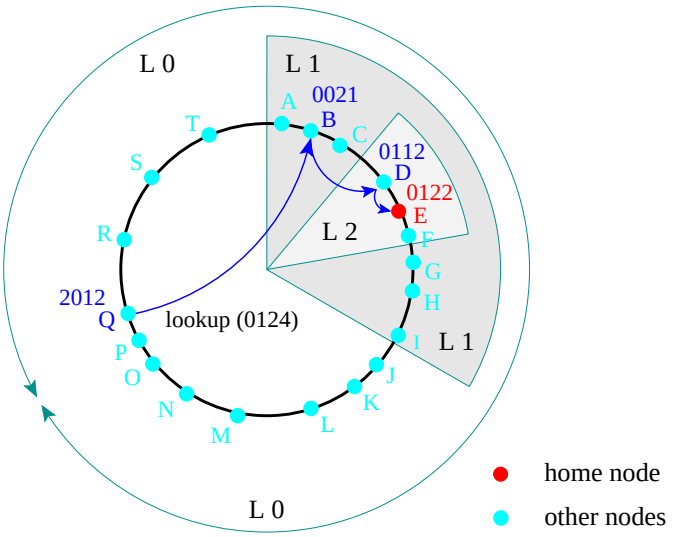

Figure 1: This figure illustrates the levels of replication in Beehive. A query for object 0124 takes 3 hops from node Q to node E, the home node of the object. By replicating the object at level 2, that is at D and F, the query latency can be reduced to 2 hops. In general, an object replicated at level i incurs at the most i hops for a lookup.

the search space is reduced exponentially, this query routing approach provides O($log_b N$) 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. 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 strives to create the minimal number of replicas such that the expected number of nodes traversed during a query will match a targeted constant, C. It uses an analytical model to derive the number of replicas required to achieve O(1) lookup performance while minimizing per node storage, bandwidth requirements and network load. We note, however, that the model is driven by estimates of object popularity and, in a real implementation like the one we describe, may deviate from the optimal due to sampling errors.

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 $log_b N$, 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 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 the M objects in the system are stored 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 $Mx_0 + \frac{M(x_1-x_0)}{b} + \cdots + \frac{M(x_k-x_{k-1})}{b^k}$. Simplifying this expression, the average per node storage requirement for replication is:

fixes with the object, but remain numerically close. This detail does not significantly impact either the time complexity of standard DHT's or our replication algorithm. Section 3 discusses the issue in more detail.

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

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 $Q(m)$ and simplifying, we arrive at 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 the second constraint 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:

$$\text{Minimize } x_0+\frac{x_1}{b}+\cdots+\frac{x_{k'-1}}{b^{k'-1}}, \text{ such that} \quad (5)$$

$$x_0^{1-\alpha}+x_1^{1-\alpha}+\cdots+x_{k'-1}^{1-\alpha} \geq k' - C', \quad (6)$$

$$\text{where } C' = C(1-\frac{1}{M^{1-\alpha}})$$

Using the Lagrange multiplier technique to solve this optimization problem, we get 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 1 hop, we obtain $k' = 2$ and the solution: $x_0 = 0.001102, x_1 = 0.0519$, 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 the case $\alpha = 1$ is as follows:

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

$$x_i^* = 1, \forall k' \leq i \leq k \quad (10)$$

$$k' \text{ given by } x_{k'-1}^* < 1$$

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, and since they consequently pay the highest per-hop latency costs as they get closer to the home node, selecting even a large target 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.

2.2 Popularity and Zipf-Parameter Estimation

The analytical model described in the previous section requires the knowledge of the parameter α of the query distribution and

the relative popularities of the objects. In order to obtain accurate estimates of the popularity of objects and the parameter of the query distribution, Beehive needs efficient mechanisms to continuously monitor the access frequency of the objects. Beehive employs a combination of local measurement and limited aggregation to keep track of the changing parameters and adapt the replication appropriately.

Each node locally measures the number of queries received by an object replicated at that node in order to estimate its relative popularity. If objects are replicated only at their home nodes, all the queries for an object are routed to the home node, and local measurement of access frequency is sufficient to estimate the relative popularity. However, if the object is replicated at level i, the queries for that object are distributed across approximately N/b^i nodes in a base b DHT with N nodes. In order to estimate the relative popularity with the same accuracy, we need an N/b^i fold increase in the measurement interval. But, this prevents the system from reacting quickly to changes in the popularity of the objects. Beehive performs limited aggregation in order to alleviate this problem and improve the responsiveness of the system.

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 access frequency of each object replicated at level i or lower and having $i+1$ matching prefixes with B. Node B receives the aggregation messages from A as well as other nodes at level i with which it shares i prefixes. It then aggregates the estimates for access frequencies received from these nodes with its own local estimate, and sends the aggregated access frequency to all nodes in the $(i+1)^{th}$ level of its routing table during the next round of aggregation . After $logN-i$ rounds of aggregation, the home node of an object replicated at level i obtains an accurate estimate of the access frequency.

In Beehive, each node is responsible for replicating an object at most one level lower. That is, nodes at level $i+1$ are responsible for replicating an object at level i. The nodes at level $i+1$ need to get the aggregated access frequencies of objects replicated at level i from the home nodes. We enable this reverse information flow by sending the aggregated access frequencies in response to aggregation messages. The home node of an object sends the latest aggregated estimate of the access frequency in response to an aggregation message from a node B. When node B receives an aggregation message from A, it sends a reply containing the aggregated access frequency of the objects listed in the aggregation message. In this manner, the access frequency of an object is aggregated at the home node and the aggregated estimate is disseminated to all the nodes containing a replica of the object. For an object replicated at level i, it takes $2(logN-i)$ rounds of aggregation to complete the information flow.

In addition to the popularity of the objects, the analytical model needs an estimate of the parameter of the query distribution. The Zipf-parameter, α, is also estimated using local measurement and limited aggregation. Each node locally computes α using the aggregated access frequency for different objects replicated at the node. We estimate α using linear regression techniques to compute the slope of the best fit line, since a Zipf-like popularity distribution is a straight line in log-scale. Since this local estimate is based on a small subset of the objects in the system, the estimate is refined by aggregating it with the local estimates of other nodes it communicates with during aggregation.

There will be fluctuations in the estimation of access frequency and the Zipf parameter due to randomness in the query distribution. In order to avoid large discontinuous changes to these estimates, we age them as follows: $estimate_\tau = estimate_{\tau-1} \times \beta + new\_value \times (1-\beta)$, with $\beta = 0.5$.

2.3 Replication Protocol

Beehive requires a protocol to replicate objects at the levels of computed 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 develop 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 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 0124 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

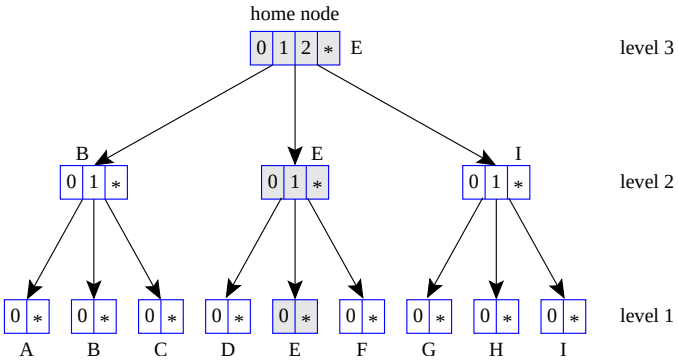

Figure 2: This figure illustrates how the object 0124 at its home node E is replicated to level 1. For nodes A through I, the numbers indicate the prefixes that match the object identifier at different levels. Each node pushes the object independently to nodes with one less matching digit.

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 0124 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 implicitly 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 for 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 within a very short time if the system is stable, that is, nodes are not joining and leaving the system.

Beehive associates a 64 bit *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 multicasts 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. Similarly, nodes joining the system but having older versions of the object replicated at them need to update the copy of their objects. We modify Beehive's replication protocol slightly to disseminate updates to nodes that have older versions due to churn in the system. During the replicate phase, each node includes the version number in addition to the object identifiers listed in the replication message. Upon receiving this message, the deciding node of an object pushes a copy of the object if it has a more recent version of the object.

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 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: A small 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 small integer used in the aggregation phase to collect and sum up the access frequencies from all dependent nodes for which this node is the deciding 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. The updates to this estimate are batched, and occur relatively infrequently compared to the query stream. 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 maintain phase, a node A sends a maintenance 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. This means that 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 maintenance messages used by Beehive and Pastry. Beehive requires some inter-node communication for replica dissemination and data aggregation. This communication is confined to pairs of nodes where one member of the pair appears in the other member's routing table. This highly stylized communication pattern suggests a possible optimization. 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 maintenance messages sent by Beehive. By piggy-backing the ith row routing table entries on to the Beehive maintenance message at replication level i, a single message can simultaneously serve as a heart beat message, Pastry maintenance message, and a Beehive maintenance 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 name (A) queries, is compatible with widely-deployed resolver libraries and is designed to provide a migration path from legacy DNS. 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 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. 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 evaluations, the Beehive maintenance interval was 48 minutes and the replication 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

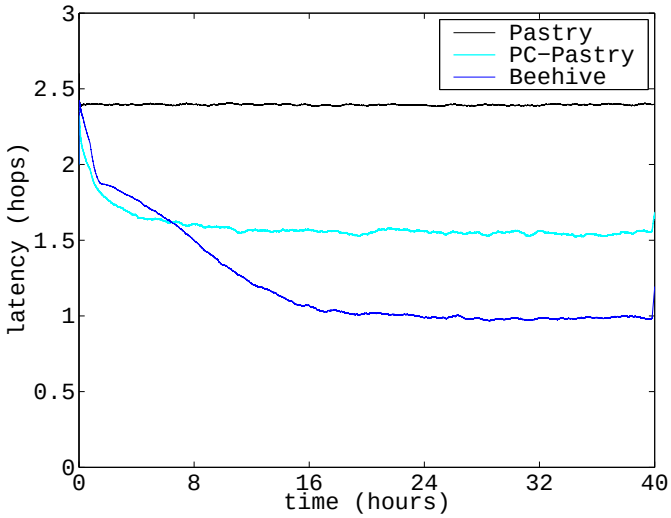

Figure 3: Latency (hops) vs Time. The average lookup performance of Beehive converges to the targeted $C = 1$ hop after two replication phases.

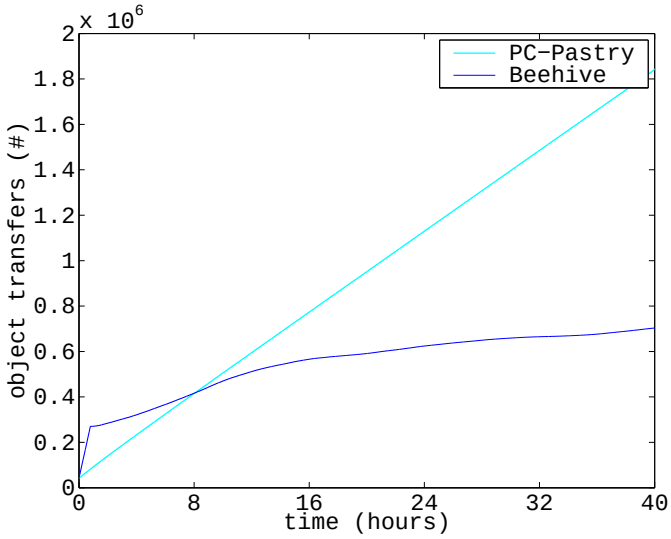

Figure 4: Object Transfers (cumulative) vs Time. The total amount of object transfers imposed by Beehive is significantly lower compared to caching. Passive caching incurs large costs in order to check freshness of entries in the presence of conservative timeouts.

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 replication phases followed by a maintain phase at each node. These three phases, combined, enable Beehive to propagate the popular objects to their respective replication levels. Once all level 1 objects have been disseminated, Beehive's proactive replication achieves 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, and 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 need to be set conservatively in order to reflect the worst case scenario under which the record may be updated, as opposed to the expected lifetime of the object. Consequently, passive caching suffers from a low hit rate as entries are evicted due to small values of TTL set by name owners.

Next, we examine the bandwidth consumed and network load incurred by PC-Pastry and Beehive for caching objects, 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 a rate of object transfer 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.

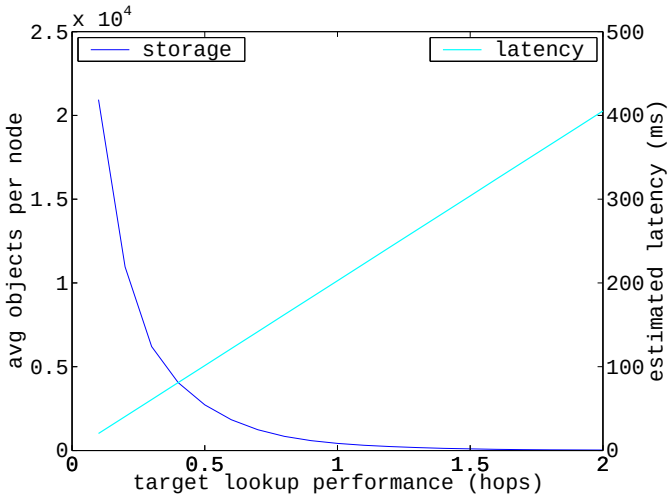

Figure 5: Storage Requirement vs Latency. This graph shows the average per node storage required by Beehive and the estimated latency for different target lookup performance. This graph captures the trade off between the overhead incurred by Beehive and the lookup performance achieved.

The average number of objects stored at each node at the end of 40 hours is 380 for Beehive and 420 for passive caching. PC-Pastry caches more objects than Beehive even though its lookup performance is worse, due to the heavy tailed nature of Zipf distributions. 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. In our large scale DNS survey, the average DNS lookup latency was 255.9 ms. Beehive with a target performance of 1 hop can provide better lookup latency than DNS.

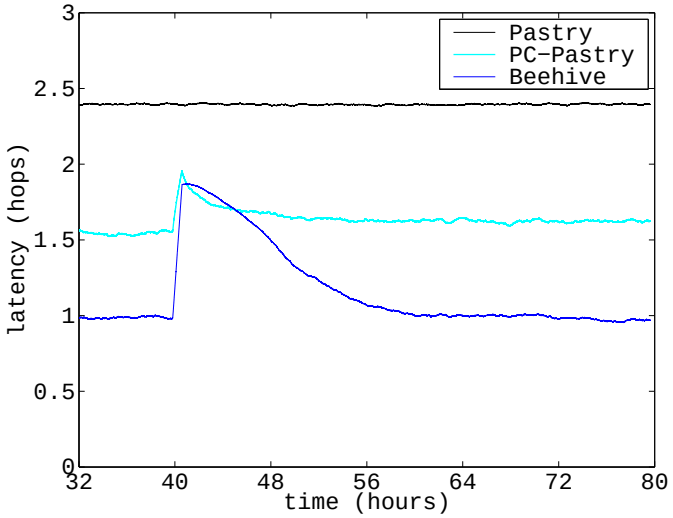

Figure 6: Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the popularity of objects and brings the average lookup performance to 1 hop.

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 for proactive replication, as objects that are least replicated suddenly need to be replicated widely, and vice versa, simulating, in essence, a set of flash crowds for the least popular objects. 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 per-

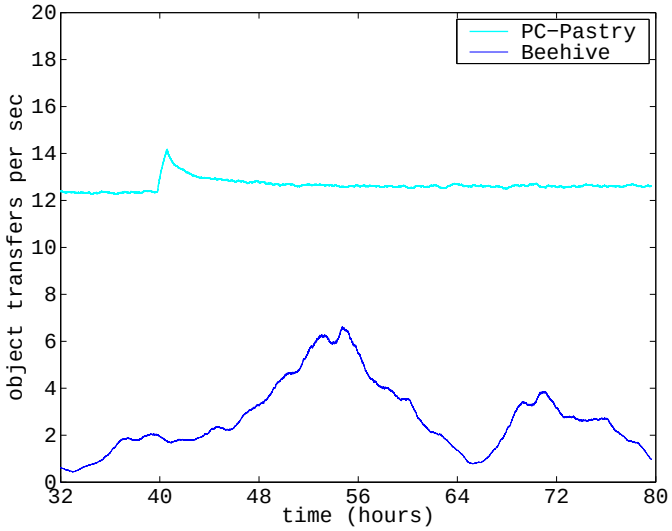

Figure 7: Rate of Object Transfers vs Time. This graph shows that when popularity of the objects change, Beehive imposes extra bandwidth overhead temporarily to replicate the newly popular objects and maintain constant lookup time.

formance of passive caching also decreases to about 1.6 hops. Figure 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.

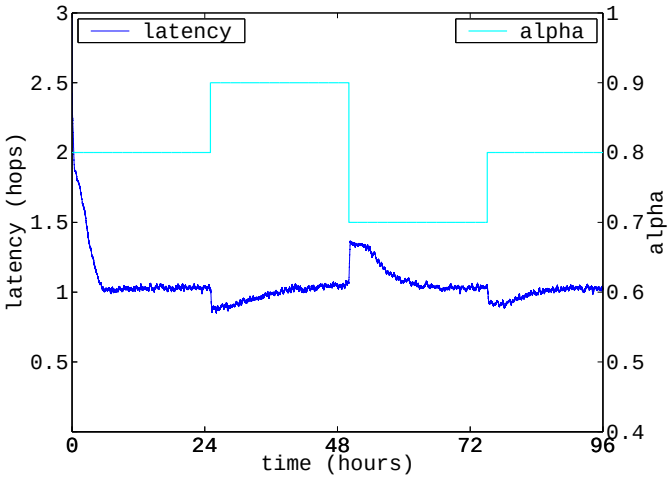

Figure 8: Latency (hops) vs Time. This graph shows that Beehive quickly adapts to changes in the parameter of the query distribution and brings the average lookup performance to 1 hop.

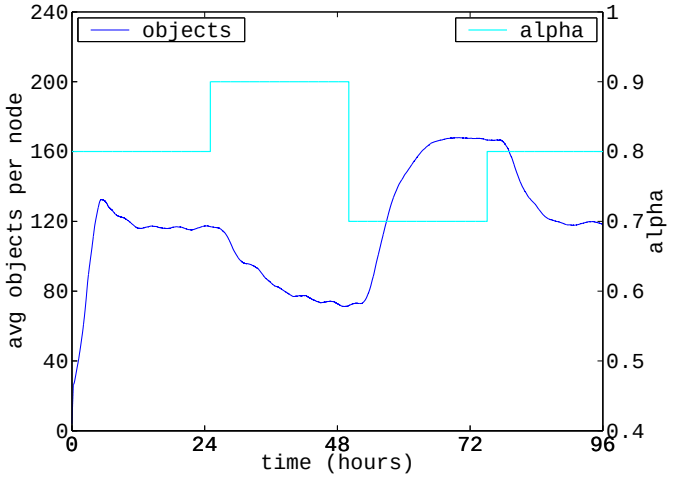

Figure 9: Objects stored per node vs Time. This graph shows that when the parameter of the query distribution changes, Beehive adjusts the number of replicated objects to maintain O(1) lookup performance with storage efficiency.

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 start with $\alpha = 0.8$, then increase it to 0.9 after 24 hours, then decrease the value to 0.7 at $t = 48$, and finally increase it to the starting value of 0.8 at $t = 72$. In order to shorten the completion time of our simulations, we performed this experiment with 4096 objects and issued queries at the rate of 4.5 queries per sec.

Figure 8 shows the lookup performance of Beehive as it adapts to changes in the parameter of the query distribution. After we started the experiment, the average query latency converges rapidly to the target of 1 hop. At 24 hours, the increase in the value of α to 0.9 causes a temporary decrease in the average query latency, but Beehive adapts to the change in the Zipf parameter and brings the lookup performance close to the target. Similarly, Beehive refines the replication levels of the objects to meet the target lookup performance, when the Zipf parameter changes to 0.7 at 48 hours and back to 0.8 at 72 hours.

Figure 9 shows the average number of objects replicated at each node in the system by Beehive. When the parameter of the query distribution is 0.8, Beehive achieves 1 hop lookup performance by replicating about 116 objects at each node on average. When Beehive observes the increase in the Zipf parameter to 0.9, it decreases the per node storage requirement to about 73 objects in order to meet the target lookup performance efficiently. Similarly, when the parameter increases to 0.7, Beehive increases the number of objects stored to about 167 per node in order to achieve the target. Overall, continuously monitoring and estimating the α of the query distribution enables Beehive to adjust the extent and level of replication to compensate for any 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

Peer to peer lookup systems proposed to date fall into two categories, namely, *unstructured* systems, where the DHT constructs an unconstrained graph among the participating nodes, and *structured* systems, where the DHT imposes some structure on the underlying network.

Unstructured peer-to-peer systems, such as Freenet [5] and Gnutella [1] perform lookups for objects using graph traversal algorithms. Gnutella uses a flooding based breadth-first-search, while Freenet uses an iterative depth-first search technique. Both Gnutella and Freenet cache queried objects along the search path to improve the efficiency of the search algorithms. However, their lookup protocols are inefficient, do not scale well, and do not provide bounds on the the average or worst case lookup performance.

Structured peer-to-peer systems are appealing because they can provide a worst-case bound on lookup performance. Several structured peer-to-peer systems have been designed in recent years. CAN [21] maps both objects and nodes on a d-dimensional torus and provides O($dn^{\frac{1}{d}}$) lookup performance by searching in a multi-dimensional space. 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 [26] use consistent hashing to map objects to nodes and route lookup requests using Plaxton's prefix-matching algorithms to search for objects. An internal database of O($logN$) entries enables these systems to route lookup requests and 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 to compute closeness of objects and nodes. Viceroy [17] provides O($logN$) lookup performance with a constant degree routing graph. De Bruijn graphs [16, 25] can achieve O($logN$) lookup performance with 2 neighbors per node and O($logN/loglogN$) with $logN$ degree per node. Beehive can be applied on any overlay based on prefix matching.

A few recently introduced DHTs provide O(1) lookup performance by tolerating increased storage and bandwidth consumption. Kelips [12] provides O(1) lookup performance with probabilistic guarantees by replicating each object on O($\sqrt{N}$) nodes. It divides the nodes into O($\sqrt{N}$) groups of O($\sqrt{N}$) nodes each and maintains information about network membership and object updates using gossip-based protocols. It maps each object to a group and replicates the object on all nodes in the group, regardless of popularity. The background gossip communication consumes a constant amount of bandwidth, but incurs long convergence time. Consequently, Kelips may not disseminate object updates to all replicas quickly. An alternative method to achieve one hop lookups is described in [13], and relies on maintaining full routing state (i.e. a complete description of system membership) at each node. The space and bandwidth costs of this approach scale linearly with the size of the network. Farsite [10] also routes in a constant number of hops, but does not address rapid membership changes. Beehive differs from these systems in three fundamental ways. First, Beehive operates as a separable layer on many DHTs without requiring structural changes. Second, it exploits the popularity distribution of objects to minimize the amount of replication. Unpopular objects are not replicated, reducing storage overhead, bandwidth consumption and network load. 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.

Several peer-to-peer applications have examined caching and replication to improve lookup performance, increase availability, and provide better failure resilience. PAST [23] and CFS [9] are examples of file backup applications built on top of Pastry and Chord, respectively. Both reserve a part of the storage space at each node to cache queried results on the lookup path and provide faster lookup. They also maintain a constant number of replicas of each object in the system in order to improve fault tolerance. These passive caching schemes do not provide any performance bounds.

Some systems employ a combination of caching with proactive object updates. In [6], the authors describe a proactive cache for DNS records. Whenever a cached DNS record is about to expire, the cache issues a fresh query to check for the validity of the DNS record, and result of the query is stored in the cache. While this technique reduces the impact of short expiration times on lookup performance, it introduces a large amount of overhead due to background object transfers, without providing bounded lookup performance.

CUP, Cache Update Propagation [20], is a demand-based caching mechanism with proactive object updates. In CUP, the process of querying for an object and updating cached replicas of that object forms a tree like structure rooted at the home node of the object. CUP nodes propagate object updates away from the home node in accordance to a popularity based *incentive* that flows from the leaf nodes towards the home node. The are several similarities between the replication protocols of CUP and Beehive. However, the decision to cache objects and propagate updates in CUP are based on heuristics, while the replication in Beehive is driven by an analytical model that enables it to provide constant lookup performance for power law query distributions.

The closest work to Beehive is [7], which presents a study of optimal strategies for replicating objects in unstructured peer-peer systems. This paper employs an analytical approach to find the best possible replication strategy for unstructured peer-to-peer systems, subject to storage constraints. The observations in this work are not directly applicable to structured DHTs, be-

cause it assumes that the lookup time for an object depends only on the number of replicas and not the placement strategy. Beehive exploits the structure of the overlay to place replicas at appropriate locations in the network to achieve the desired performance level.

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 level, C, to a lower, fractional value. 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 unique properties desirable for a large class of applications, including self-organization, failure resilience, high scalability, and a worst-case performance bound. However, their O($logN$) average-case performance has prohibited them from being deployed for latency-sensitive applications, including DNS. In this paper, we outline a framework for proactive replication that can improve the average-case lookup performance of prefix-based DHTs to $O(1)$ for a frequently encountered class of query distributions.

The Beehive framework consists of three components, layered on top of a standard DHT substrate, such as Pastry. An analytical model provides a closed form solution for computing the requisite level of replication in order to achieve a targeted lookup performance. This analytical solution is optimal in the number of replicas for Zipf-like distributions with $\alpha \leq 1$. An estimation technique, based on local measurements and limited aggregation to address statistical fluctuations, derives input parameters for the model. The estimation process is integrated with background traffic already present in the DHT. Computing the level of replication for each object is performed independently at each node, without costly consensus or synchronization. A replication algorithm proactively disseminates the objects throughout the system, along the routing tables already maintained by the underlying DHT.

Analysis of Beehive's performance in the context of a DNS application indicates that it can achieve a targeted performance level with low overhead. Beehive adapts quickly to flash crowds, which can alter the relative popularities of the objects in the system. It detects qualitative shifts in the global query distribution and adjusts replication parameters accordingly to compensate. The implementation is small and the Beehive approach can be applied to other latency-sensitive applications. Overall, the system derives its efficiency by taking advantage of the underlying structure of the lower-layer DHT, and makes it feasible to use DHTs in low-latency applications where the query distribution follows a power law by decoupling lookup performance from the size of the network.

References

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

[2] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. "Web Caching and Zipf-like Distributions: Evidence and Implications." *IEEE International Conference on Computer Communications (INFOCOM) 1999*, New York NY, March 1999.

[3] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach. "Secure Routing for Structured Peer-to-Peer Overlay Networks." *Symposium on Operating Systems Design and Implementation, OSDI 2002*, Boston MA, December 2002.

[4] Miguel Castro, Peter Druschel, Charlie Hu, and Antony Rowstron. "Exploiting Network Proximity in Peer-to-Peer Overlay Networks." *Technical Report MSR-TR-2002-82, Microsoft Research*, May 2002.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." *Lecture Notes in Computer Science*, vol 2009, pp 46-66, 2001.

[6] Edith Cohen and Haim Kaplan. "Proactive Caching of DNS Records: Addressing a Performance Bottleneck." *Symposium on Applications and the Internet SAINT 2001*, San Diego-Mission Valley CA, January 2001.

[7] Edith Cohen and Scott Shenker. "Replication Strategies in Unstructured Peer-to-Peer Networks." *ACM SIGCOMM 2002*, Pittsburgh PA, August 2002.

[8] Russ Cox, Athicha Muthitacharoen, and Robert Morris. "Serving DNS using a Peer-to-Peer Lookup Service". *International Workshop on Peer-To-Peer Systems 2002*, Cambridge MA, March 2002.

[9] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. “Wide-area cooperative storage with CFS.” *ACM Symposium on Operating System Principles SOSP 2001*, Banff Alberta, Canada, October 2001.

[10] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, Marvin Theimer. “Reclaiming Space from Duplicate Files in a Serverless Distributed File System.” *International Conference on Distributed Computing Systems, ICDCS 2002*, Vienna, Austria, July 2002.

[11] D. Eastlake. “Domain Name System Security Extensions”. *Request for Comments (RFC) 2535*, 3^{rd} ed., March 1999.

[12] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robert van Rennesse. “Kelips: Building an Efficient and Stable P2P DHT Through Increased Memory and Background Overhead.” *Second International Peer-To-Peer Systems Workshop, IPTPS 2003*, Berkeley CA, February 2003.

[13] Anjali Gupta, Barbara Liskov, Rodrigo Rodrigues. “One Hop Lookups for Peer-to-Peer Overlays.” *Ninth Workshop on Hot Topics in Operating Systems*. Lihue, Hawaii, May 2003.

[14] Nicholas Harvey, Michael Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. “SkipNet: A Scalable Overlay Network with Practical Locality Properties.”, *Fourth USENIX Symposium on Internet Technologies and Systems, USITS 2003*, Seattle WA, March 2003.

[15] Jaeyon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. “DNS Performance and Effectiveness of Caching.” *ACM SIGCOMM Internet Measurement Workshop 2001*, San Francisco CA, November 2001.

[16] Frans Kaashoek and David Karger. “Koorde: A Simple Degree-Optimal Distributed Hash Table.” *Second International Peer-To-Peer Systems Workshop, IPTPS 2003*, Berkeley CA, February 2003.

[17] Dahlia Malkhi, moni Naor, and David Ratajczak. “Viceroy: A Scalable and Dynamic Emulation of the Butterfly.” *ACM Symposium on Principles of Distributed Computing, PODC 2002*, Monterey CA, August 2002.

[18] Petar Maymounkov and David Maziéres. “Kademlia: A Peer-to-peer Information System Based on the XOR Metric.” *First International Peer-To-Peer Systems Workshop, IPTPS 2002*, Cambridge MA, March 2002.

[19] Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa. “Accessing nearby copies of replicated objects in a distributed environment.” *Theory of Computing Systems*, vol 32, pg 241-280, 1999.

[20] Mema Roussopoulos and Mary Baker. “CUP: Controlled Update Propagation in Peer-to-Peer Networks.” *USENIX 2003 Annual Technical Conference*, San Antonio TX, June 2003.

[21] Sylvia Ratnasamy, Paul Francis, Mark Hadley, Richard Karp, and Scott Shenker. “A Scalable Content-Addressable Network.” *ACM SIGCOMM 2001*, San Diego CA, August 2001.

[22] Antony Rowstorn and Peter Druschel. “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems.” *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, Heidelberg, Germany, November 2001.

[23] Antony Rowstorn and Peter Druschel. “Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility.” *ACM Symposium on Operating System Principle, SOSP 2001*, Banff Alberta, Canada, October 2001.

[24] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. “Chord: A scalable Peer-to-peer Lookup Service for Internet Applications.” *ACM SIGCOMM 2001*, San Diego CA, August 2001.

[25] Udi Wieder and Moni Naor. “A Simple Fault Tolerant Distributed Hash Table.” *Second International Peer-To-Peer Systems Workshop, IPTPS 2003*, Berkeley CA, February 2003.

[26] Ben Zhao, Ling Huang, Jeremy Stribling, Sean Rhea, Anthony Joseph, and John Kubiatowicz. “Tapestry: A Resilient Global-scale Overlay for Service Deployment.” *IEEE Journal on Selected Areas in Communications, JSAC*, 2003.