Leopard: A Locality Aware Peer-to-Peer System with No Hot Spot*

Yinzhe Yu, Sanghwan Lee, and Zhi-Li Zhang

Department of Computer Science and Engineering,
University of Minnesota,
Minneapolis MN 55455, USA
{yyu, sanghwan, zhzhang}@cs.umn.edu

Abstract. Recent research [7, 12, 2] has shown that Internet hosts can be efficiently (i.e., without excessive measurements) mapped to a virtual (Euclidean) coordinate system, where the geometric distance between any two nodes in this virtual space approximates their real IP network distance (latency). Based on this result, in this paper, we propose an alternative approach that *inherently* incorporates a virtual coordinate system into a P2P network. In our system, called Leopard, a node is assigned a coordinate in the so-called *node geo space* as it joins the network, and obtains neighbor relationships that reflects network proximity from the beginning. The object id space and the node geo space are then "weaved" together via a novel technique called *geographically-scoped hashing.* Through analysis and simulation, we show three major desirable properties of Leopard to exemplify the power of this paradigm shift: i) a constant routing stretch, i.e., IP level network latency of object look-up is proportional to the distance between a requesting node and the target object; ii) always locates a near-by copy when multiple copies exist; and iii) effectively handles "flash crowd" traffic with near optimal load balancing.

Keywords: peer-to-peer, lookup service, virtual coordinates, locality-awareness.

1 Introduction

A fundamental challenge in Peer-To-Peer (P2P) systems is how to locate objects of interest in the network, namely, the *look-up service.* A key break-through towards a scalable and distributed solution of the lookup problem is the *distributed hash table (DHT)* (including Chord[9], CAN[11], Tapestry[14] and Pastry[10], among others). However, since both object (any entities of interest) id and network node are *randomly hashed* to a same id space, "locality-awareness" is *not inherent* in the basic design of DHT. As a result, *routing stretch* of object look-up, defined by the ratio of the network distance traveled by a look-up query message

* This work was supported in part by NSF grant ITR-0085824 and CNS-0435444.

and the distance between the requesting node and the nearest copy of the target object, can be high. Another challenging problem is to effectively cope with the so-called "flash crowd" or "hot spot" problem, namely, a sudden surge of user requests for a popular object. Since in standard DHT an object is randomly hashed to a single id (or a few id's, if replicated using several hash functions) in the id space, the node(s) responsible for storing or answering queries for that object can be overwhelmed by a "flash crowd," creating a "hot spot" in the system.

Recent research [7, 12, 2] has shown that Internet hosts can be efficiently (i.e., without excessive measurements) mapped to a virtual (Euclidean) coordinate system, such that the geometric distance between any two nodes in the virtual space accurately approximates their real IP network distance (latency). For example, [7] shows that with an 8-dimension virtual space, more than 90% of inter-nodal distances in the virtual space are within 50% error margin of their real network distances. One of the intended usage of such a virtual coordinates system is to improve the neighbor relationship in the P2P overlay. For example, a node can gradually select close-by fingers (routing neighbors) in the virtual space to reduce routing stretch. However, such *incremental* change to existing scheme may not be the best way to maximize the utility of this new facility.

In this paper, we propose to inherently incorporate locality-awareness into a P2P system. We separate the *object id space* from the *node space*: each object is assigned a unique id in an object id space; while each node is assigned a *coordinate* in a so-called *(node) geo space*, as it joins the network. During the join process, the node obtains neighbor relationships that reflects "network proximity" *from the beginning.* The object id space and the node geo space are then "weaved" together via a technique called *geographically-scoped hashing* (GSH): each object id is mapped into multiple points (i.e., coordinates) in the node geo space with varying geographical scopes; the nodes that are closest to these points are responsible for maintaining "pointers" to the object and performing look-up queries for it. Through analysis and simulations, we demonstrate that: i) Leopard has a constant IP-level routing stretch; this holds not only in an *average* or *probabilistic* sense, but also in the *worst-case.* ii) Leopard always locates a *near-by* copy when multiple copies exist. iii) Leopard effectively handle "flash crowd" with near optimal load balancing, without adding *exogenous* complexity in look-up operations.

In the remainder of the paper, we first present an overview of our scheme in Section 2. The design details of Leopard are described in Section 3 and 4, focusing on object operations and node space management respectively. Performance evaluation results based on packet level simulations are included as Section 5. Finally, we briefly compare our work with related works in Section 6 before concluding in Section 7.

2 Key Concepts: Area Hierarchy and GSH

Same as standard DHT, we assign each object a unique *identifier* from an *id space*, based on either application semantics or random hashing. Without loss

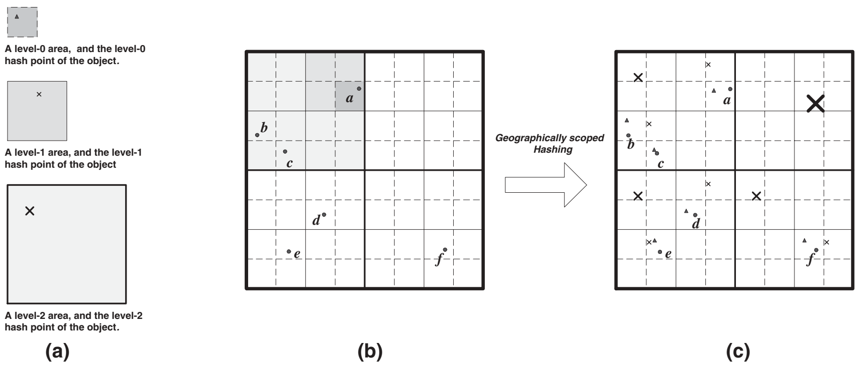
A level-0 area, and the level-0 hash point of the object.
A level-1 area, and the level-1 hash point of the object
A level-2 area, and the level-2 hash point of the object.
(a)
a
b
c
d
e
f
(b)
Geographically scoped Hashing
(c)

Fig. 1. An example of geographically-scoped hashing. (a) An object is hashed to different relative points in different levels of areas. (b) Locations of six owners ($a - f$) of a same object ω. Note that the level-$0, 1, 2$ areas of node a are highlighted with different shades. (c) Hash points (pointer nodes) of the six owners at various levels of area

of generality, we assume that the object id is a scalar (e.g., a 128-bit number). We use $\omega.ID$ to denote the identifier of an object ω. On the other hand, nodes form the *node geo space* – a finite d-dimensional metric space – upon which a coordinate system is defined. A nodes's *coordinate* is determined when it joins the system. The coordinate can be the node's actual geographical location obtained via GPS, or a virtual coordinate obtained via a virtual coordinates service such as GNP [7], Virtual Landmark [12] or Vivaldi [2]. For simplicity of exposition, we assume that the node geo space is a d-dimensional Euclidean space, with inter-nodal distance approximating IP level network distance.

We introduce a *hierarchical grid* over the node geo space by dividing it into a hierarchy of (d-dimensional) *areas*: at the highest level, the entire space is the level-L area, where L is a system parameter specifying the total number of levels in the hierarchy. For $1 \leq l \leq L$, each level-l area is divided into 2^d level-$(l-1)$ areas, obtained by cutting the level-l area into half along each of the d dimensions. Fig.1(b) illustrates such a hierarchy of areas with $L = 3, d = 2$, where square areas of different levels are delineated with different line styles. For $l = 0, 1, \ldots, L$, let A_l be the level-l area containing a node a. In Fig. 1(b), we use four different levels of shades to show $A_0 \ldots A_3$. We see that $A_l \subset A_{l+1}, l = 0, ..., L-1$, and A_L is the entire node geo space. Let r_l denote the *size* of a level-l area A_l (i.e., its side length), then $r_{l+1} = 2r_l$. We will use $\mathcal{O}(A_l)$ to denote A_l's *origin* coordinate, i.e., point in A_l with the smallest coordinates.

We now introduce the concept of *geographically scoped hash* functions. For $l = 0, 1, \ldots, L$, let $\mathcal{H}_l$ be a d-dimensional random hash function[1] with the range $[0, r_l)^d$. Given an object ω and a level-l area A_l, the *hash point* of ω under (the geographical scope) A_l is a point within A_l given as $\mathcal{H}(\omega, A_l) = \mathcal{O}(A_l) +$

[1] A d-dimensional hash function can be constructed as a Cartesian product of d independently 1-dimensional random hash functions.

$\mathcal{H}_l(\omega.ID)$. In Fig. 1(a), we illustrate the hash functions for three areas (scopes) of different levels. Fig. 1(b) shows the locations of six nodes $(a - f)$, each owning a copy of the object ω. Fig. 1(c) shows the hash points of the six nodes in various areas. The node in the node geo space that is *closest*[2] to the hash point $\mathcal{H}(\omega, A_l)$ of object ω is referred to as the *level-l pointer node* for object ω in area A_l, and is denoted by $\mathcal{P}(\omega, A_l)$. Pointer nodes of an object are responsible for maintaining object "pointers" and answering look-up queries.

We now give a high-level illustration on how these hash points are used for locating objects in Leopard. When a node a wishes to share object ω, it *publishes* ω by "planting" a pointer at pointer node $\mathcal{P}(\omega, A_0)$. For $l = 1, \ldots, L-1$, each level-l pointer node $\mathcal{P}(\omega, A_l)$ in turn computes the next-level hash point $\mathcal{H}(\omega, A_{l+1})$ and plants a pointer at $\mathcal{P}(\omega, A_{l+1})$. Now suppose a node b (let B_l denote its level-l areas) is interested in ω. It first sends a look-up query to $\mathcal{P}(\omega, B_0)$. Note that if a and b reside in the same level-0 area, then $\mathcal{P}(\omega, B_0)$ (=$\mathcal{P}(\omega, A_0)$) will be able to direct node b to node a for the object. Otherwise, $\mathcal{P}(\omega, B_0)$ computes the level-1 hash point and forwards the query to $\mathcal{P}(\omega, B_1)$. The process goes on recursively. If nodes a and b reside in the same level-l area, i.e., $A_l = B_l$, then the level-l pointer node $\mathcal{P}(\omega, B_l)$ $(= \mathcal{P}(\omega, A_l))$ will have a pointer to object ω, and thus can direct node b to node a for object ω. As $A_L = B_L$, in at most L steps, node b will be able to locate a pointer to object ω. In a sense the pointer nodes of an object ω form a distributed search tree (embedded in the node geo space), where each edge connects $\mathcal{P}(\omega, A_l)$ to $\mathcal{P}(\omega, A_{l+1})$, as shown in Fig. 2(a).

3 Leopard Look-Up Service

In Leopard *pointers* – information about objects – are stored in various pointer nodes. To reduce the storage cost in pointer nodes and to shield high level pointer nodes from frequent publish/withdraw operations in the large area it is in charge, *Leopard only maintains precise information (the IP address) of individual object owners at level-0 pointer nodes.* Each level-0 pointer node of an object ω has an *owner list* that lists all the object owners of ω in this level-0 area. In higher-level pointer nodes Leopard maintains *aggregate* information: a `TRUE/FALSE` value (called a *branch indicator*) for each of its 2^d next lower level areas, indicating whether that area contains at least one object owner. By following a series of branch indicators with `TRUE` values down a branch of the object search tree the IP address of an object owner can be obtained at the level-0 pointer node (leaf of the tree).

Fig. 2(a)(left) depicts an object search tree with $d = 2$ and $L = 3$, for five object owners $a - e$, located in positions as shown in Fig. 1. A black node represents an *active* pointer node, a pointer node currently storing a pointer to the object. A white node represents an *inactive* pointer node — a node closest to a hash point in the area, but stores no pointer because there is no object owner

[2] We will give a precise definition of *closeness* in Section 4.

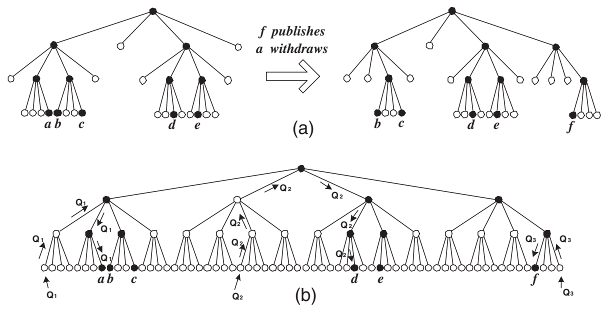

Fig. 2. (a) Initially, a object search tree (left) have five owners (a, b, c, d, and e). After f publishes and a withdraws, the object search tree becomes the one on the right. (b) Three examples (Q_1, Q_2, Q_3) of query message path

in that area yet. Each active pointer node has $2^d = 4$ children. An active child corresponds to a `TRUE` branch indicator and an inactive child corresponds to a `FALSE` branch indicator. Note that not all the inactive pointer nodes are shown in Fig. 2(a) (e.g., children of an inactive pointer node).

To publish an object ω, an owner a in a level-0 area A_0 first computes the hash point $\mathcal{H}(\omega, A_0)$, and sends a publish request to its level-0 pointer node $n := \mathcal{P}(\omega, A_0)$[3]. If n is currently *active*, i.e., it already has a pointer for ω, it simply appends node a's IP address and coordinate to the object owner list in the existing pointer. Otherwise, it creates a new level-0 pointer, and notifies the level-1 pointer node $\mathcal{P}(\omega, A_1)$ that its area now contains an owner of ω. This process continues recursively until either an active pointer node or the root (top-level) pointer node is reached. During the process, the corresponding branch indicators at pointer nodes in every levels are set accordingly. Algorithm 1 list the steps of a publish operation. It can be viewed as a recursive RPC(remote procedure call). When a new owner a want to publish an object ω, it simply calls $n.\mathbf{publish}(\omega, 0, a.coord, a.IP)$, where $n := \mathcal{P}(\omega, A_0)$.

The object withdraw operation involves the similar recursive process to adjust the object owner list and the branch indicators in pointers of various level of pointer nodes. Figure 2(a) shows an example where node f publishes and node a withdraws. The example shows that by storing the aggregate information of "whether a branch contains an owner of ω or not," high-level pointer nodes are often not affected when nodes publish and withdraw.

When a node x in a level-0 area X_0 wants to look up for an object ω, it sends a query for object ω to the pointer node $\mathcal{P}(\omega, X_0)$. If the pointer node contains a level-0 pointer for ω, it replies to x with the IP address of an owner. Otherwise, the pointer node recursively queries the higher-level pointer nodes, until a pointer to object ω is found. Starting at that pointer node, the query is sent down to

[3] As we will show in Section 4, by including a target point in the packet header and performing greedy forwarding, a packet can be sent from any source node towards any point (e.g., a hash point) in the node geo space.

Algorithm 1 : n.**publish**$(\omega, l, a.coord, a.IP)$ `//obj,level,coordinate,IP addr`

```
//lookup the pointer database for an entry with the three-field key
$entry \Leftarrow$ lookupPointer$(\omega, l, \mathcal{H}(\omega, A_l))$
if $l = 0$ then
    //always create a new pointer and append it to owner list
    storePointer$(\omega, l, \mathcal{H}(\omega, A_l), a.coord, a.IP)$
    if $entry$ = NULL then
        //publish at higher level recursively
        $n' \Leftarrow \mathcal{H}(\omega, A_{l+1})$
        $n'$.publish$(\omega, l+1, a.coord, n.IP)$
else
    if $entry$ = NULL then
        //create a new pointer only if it's the first owner in the area
        $entry \Leftarrow$ storePointer$(\omega, l, \mathcal{H}(\omega, A_l))$
        $n' \Leftarrow \mathcal{H}(\omega, A_{l+1})$
        $n'$.publish$(\omega, l+1, a.coord, n.IP)$
    //determine branch indicator index using level l and a's coordinate
    $br\_id \Leftarrow$ getBranchID$(l-1, a.coord)$
    //set branch indicator accordingly
    $entry.branch[br\_id] \Leftarrow$ TRUE
return
```

the lower-level pointer nodes by following a branch where the branch indicators are TRUE[4], until a level-0 pointer node is reached. The level-0 pointer node then replies node x with the IP address of an owner. Due to space limitation, we refer the reader to the technical report version of this paper [13] for a detailed list of Leopard Query algorithm. Figure 2(b) illustrates three examples (Q_1, Q_2 and Q_3) of object query paths.

As shown in the example, when an owner a and a querying node x reside in a common level-l area A_l, the query can be performed in $2l$ (logical) steps in the tree. In fact, the distance traveled by a query message is also bounded by $O(r_l)$, where r_l is the size (side length) of a level-l area, as stated in the following theorem[5]:

Theorem 1. *Suppose a node x queries for an object ω in Leopard, and the located object owner shares a level-l area with x. The total geometric distance traveled by the query message (summing up all the $2l$ steps) in the node geo space is bounded in worst case by $4\sqrt{d}r_l$, where r_l is the size of a level-l area.*

Note that Theorem 1 gives a *worst-case* bound on the geometric distance traveled by a query message from the requester to the *located owner*. Leopard can also be optimized (using what we call "sibling indicators" as described in [13]) so that the

[4] When there are multiple TRUE branch indicators, the pointer node can either choose one of them randomly, or employ certain scheduling strategy, such as round robin.

[5] Due to space limit, we refer [13] for all proof of theorems in this paper.

located owner is near-optimal (by a constant factor) compared with the optimal owner (closest to the requester). In particular, we have the following theorem.

Theorem 2. *Suppose a node x is querying for an object ω. Let the owner located by Leopard be s_1, and the closest owner in the network be s_2. Let $D(a, b)$ denote the distance between two points a and b in the geo space. Then we have either $D(x, s_1) - D(x, s_2) \leq 2\sqrt{d}r_0$ or $\frac{D(x,s_1)}{D(x,s_2)} \leq 4\sqrt{d}$.*

We note that that from Theorems 1 and 2, the distance traveled by a query message to locate an object in Leopard is at most a constant factor of the optimal distance, i.e., the geometric distance between the requester and the closest object owner in the node geo space. If we assume that the distance in the node geo space accurately approximates IP level network distance, then Leopard achieves a constant routing stretch at IP level even in the worst case.

3.1 Mitigating Hot Spots

To cope with the "flash crowd" problem, Leopard imposes the following simple rule on nodes requesting for an object:*A node x starting the transfer(downloading) of an object ω from another node (located through Leopard) must publish ω for at least the duration of the object transfer; in addition, x must honor any transfer request accepted during its publishing period.* This rule creates an object propagation model similar to the popular Bit-Torrent system. However, since Leopard always returns a nearby copy of an object, it not only releases the "hot spot" on the hosting nodes (as also achieved by Bit-Torrent), but also greatly reduces the routing cost of the "flash crowd", since each request is resolved "locally." The object search tree embedded in the node geo space achieves good load balance naturally without maintaining a complex object tracker, as Bit-Torrent does.

We next show that based on this rule, Leopard effectively mitigates the "flash crowd" problem with excellent load balancing. We measure the balance of load with two metrics. The first metric, *owner service count*, is defined as the number of object transfer requests an object owner serves during its publishing time (assuming it withdraws immediately after finishing downloading itself). Ideally, this metric can be as low as 1 even with extremely high query rate, i.e., each requesting node can serve the next requester. The second metric, *pointer node service count*, is the number of query messages a pointer node of ω serves during its downloading period of ω. We have the following theorem regarding the upper bound of the two metrics.

Theorem 3. *i) The number of object transfer requests an object owner a serves during its publishing period is bounded by $L + n_0$, where n_0 is the number of nodes in the level-*0 *area A_0. ii) The number of queries a pointer node a serves during* one downloading period *is bounded by $L + n_0$ (for level-*0 *pointer node) or $L + 2^d$ (otherwise).*

Since we have $L = O(\frac{\log{(N/n_0)}}{d})$ (N being total node number), the two metrics are bounded by $O(\frac{\log N}{d})$ and $O(\frac{\log N}{d} + 2^d)$ respectively. Therefore, Theorem 3 guarantees that *regardless of the object request rate*, the two metrics grow proportional to $\frac{\log N}{d}$.

4 Node Space Management

In Leopard each node a is not only assigned a d-dimensional coordinate, (denoted by $(x_1^a, \ldots, x_d^a)$), but is also responsible for a portion of the node space around its coordinate called a *zone*. The node geo space is divided into zones that satisfy the following two properties: 1) the boundaries of a zone are hyperplanes perpendicular to axis of dimensions; and 2) a node's zone always contain its coordinate. A node a's zone, denoted by Z_a, can be represented by d ranges $\{[u_1^a, v_1^a), [u_2^a, v_2^a), \ldots, [u_d^a, v_d^a)\}$, such that $u_i^a \leq x_i^a < v_i^a, \forall 1 \leq i \leq d$. A *neighbor* zone of Z_a is a zone adjacent to it on a boundary. Formally, Z_b is a neighboring zone of Z_a if $\exists i \in \{1, \ldots, d\}$ such that $u_i^a = v_i^b$ or $v_i^a = u_i^b$, and $\forall j \in \{1, \ldots, d\} \setminus \{i\}, [u_j^a, v_j^a) \cap [u_j^b, v_j^b) \neq \Phi$, i.e., there exists a dimension where the two zones are adjacent, for the remaining dimensions, the two zones overlap. Each node maintains a *neighbor table* about neighbor zones: the ranges of neighbor zones and the coordinate and IP address of the nodes owning them. Zones are dynamically created and re-structured as nodes join and leave the node geo space. Note also that the zone structure is independent of the area hierarchy defined earlier.

4.1 Greedy Forwarding in Node Space

As we mentioned earlier, the actual packet forwarding in Leopard is based on a greedy algorithm guided by a destination coordinate stamped in the packet header. To forward a packet to a destination point p in node geo space, every intermediate node simply forward it to the neighbor with a zone *closest* to p, until the current node is already closest (than any neighbor), in which case the current node is the destination. To measure the *closeness* between a zone and a point in the node geo space, we define the distance $\mathcal{D}(Z_a, p) = \min\{d_q | \forall q \in Z_a, d_q = ||q - p||\}$, i.e., the distance between the closest point in the zone (to p) and the point p. The rationale behind this definition of distance is two folded: 1) it's easy to calculate[6]; 2) based on this definition, our greedy forwarding algorithm ensures delivery of packet, as guaranteed by Theorem 4. Theorem 4 rules out the possibility of the *local minima* problem associated with many greedy algorithm, and guarantees delivery of packet in finite number of steps.

Theorem 4. *With the above definition of distance between a zone Z_a and a point p in the node space, a node always has a neighbor closer to p, unless it is the destination node ($p \in Z_a$).*

[6] Let q^* denote the closest point in Z_a (to p), we can obtain its coordinates as follows.

$$x_i^{q^*} = \begin{cases} x_i^p \text{ if } x_i^p \in [u_i^a, v_i^a) \\ u_i^a \text{ if } x_i^p < u_i^a \\ v_i^a \text{ if } x_i^p \geq v_i^a \end{cases}$$

Therefore, $\mathcal{D}(Z_a, p) = \mathcal{D}(q^*, p)$. Fig. 3(a) shows the corresponding closest point q^*'s from three different points p_1, p_2 and p_3 to a rectangular zone.

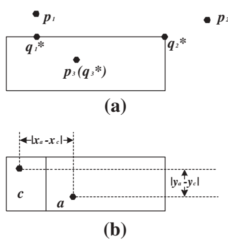

Fig. 3. (a) Determine closest point in a zone to a target point. (b) Zone splitting example

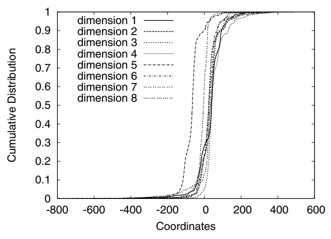

Fig. 4. Cumulative distribution of coordinates in a GNP data set

4.2 Node Join and Leave

When a node a wants to join Leopard, it follows the procedure below.

1. a obtains its coordinates through an external service such as GNP.
2. a finds a *bootstrap* node b in Leopard, through a well-known rendezvous point (e.g., a URL).
3. a sends out a `Join Request` message through the bootstrap node b, carrying a's coordinate as the destination. Using the greedy forwarding algorithm, the `Join Request` message will arrive at the node currently owning a zone that contains a's coordinate, denoted by c.
4. a and c split the zone. To do so, they first choose a dimension in which a and c's coordinates have the largest difference. They then divide the zone into two parts at the mid-point along the chosen dimension. Each part serves as one node's new zone. Figure 3(b) shows an example of zone splitting.
5. After obtaining new zones, a and c each updates its neighbor table. Note that both a and c's new neighbor sets are subsets of node c's original neighbor set.

When a node a leaves the network gracefully, it performs the following procedure.

1. a withdraws all objects it is currently publishing.
2. a notifies its neighbors of its intention to withdraw. Those neighbors with zones *mergable* with a's zone will take over, appropriately dividing a's zone into sub-zones if necessary, so that they can be merged with their own zones. (The sub-division of the zone is necessary to maintain the zone properties postulated at the beginning of this section.)
3. a transfers all the state information (pointers) to those nodes that are taking portion of its zone.

5 Simulation Experiments

We vary three key parameters in our packet level simulation experiments: number of dimensions d, number of levels L, and the node distribution in geo space. We

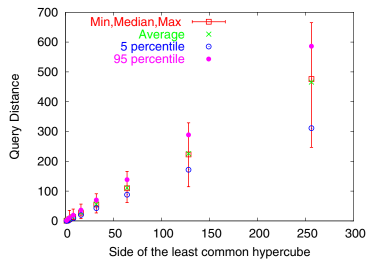

Fig. 5. Query distance vs. smallest common area size. d=2, L=8, uniform

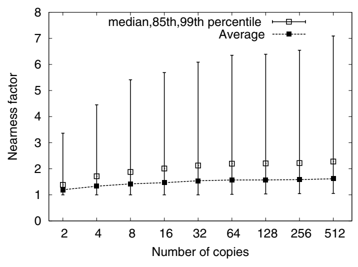

Fig. 6. Nearness factor vs. number of object copies. $d = 2, L = 8$, uniform

use $d = 2$ and 8 as $d = 2$ reflects the actual geographical coordinates; while $d = 8$ is shown [7, 12] to be enough for accurate approximation. The hierarchy level L is determined such that the number of nodes in level-0 areas does not exceed a small constant. In addition to the simple uniform node distribution, we generate node distributions that model realistic Internet nodes to address the concern of pointer node load balancing. This is done by using the coordinate distributions of the GNP data sets [1]. Fig. 4 shows the coordinate distribution on each of the 8 dimensions for the 869 virtual coordinates generated by GNP.

Query Distance. To verify Leopard *routing stretch*, we define *query distance*, computed by summing up the geometric distances in node geo space of each forwarding hop of a query message. Provided that the virtual coordinate system is accurate, query distance is a reliable indicator of real network distance. We construct a network of 10^5 uniformly distributed nodes, and distribute $1,000$ objects into random nodes such that the i^{th} $(1 \leq i \leq 1000)$ object has i copies. We generate $100,000$ queries from random node for a random target object, and record the query distances for every query. Fig. 5 shows the distribution of the query distances grouped by the *smallest common areas* of the querying nodes and the *located* owner. The x-axis in Fig. 5 represents the size (i.e., side length) of the smallest common area, normalized by r_0. We observe a clear linear relation between the query distance and the common area size, which strongly support our analysis results about Leopard's constant routing stretch. In addition, the actual value of that "constant stretch" is small: less than 2 for average, and less then 2.5 for 95 percentile.

Locating Nearby Copies. We first define a metric called *nearness factor*, which is the ratio of the distance from the querying node to the *located* owner and that distance to the actual *nearest* owner in the network. We construct a system $(d = 2, L = 8)$ with 10^5 uniformly distributed nodes and 100 unique objects, each with 2^k copies in the network (we vary the value of k from 1 to 9 in nine different experiments). For each experiment, we generate 5000 queries from random nodes for a random object. Fig. 6 shows the statistics of nearness factors. The average, median, 85th percentile and 99th percentile are computed for the nine experiments with different k's. We see that the nearness factor is small: the *medians* are 1 for all k values, indicating at least half of the querying nodes being able to find the actual *nearest* copy.

Flash Crowd Mitigation. We generate queries to a *single* popular object with different query rates and measure the object owner service count (OSC) and pointer-node service count (PSC). The simulation starts with a 10^5 node network, and a randomly chosen node publishes the target object. Random queries are then generated with a Poisson distribution with a mean of 2^q queries per second ($q = 0 \ldots 6$). We performs five runs of the experiment. Each run lasts $1,000$ seconds, i.e., for $q = 6$ per second, $64,000$ nodes obtain a copy. Since we fix object downloading time at 100 seconds, and node always withdraws immediately after downloading, the system maintains 100×2^q copies in steady state. Two different sets of system parameters are used: i) $d = 2, L = 8$, uniform node distribution and ii) $d = 8, L = 13$ with non-uniform distribution. Table 1 shows the histograms (accumulated over five runs) of nodes with different OSCs. For example, when query rate is 1/s (uniform), a total of 4991 ($2654 + 2 \times 948 + 3 \times 133 + 4 \times 8 + 5 \times 2$) requests are served in five runs. Only two nodes ever served five requests. For all query rates, vast majority of nodes (99.5%+ and 99.9%+ in two distributions) serve three requests or less. Based on Table 1, we plot the fraction of nodes has OSCs of 1 through 5, as shown in Fig.7. We observe that the fractions are almost identical, indicating Leopard's capability to achieve near-optimal load balancing regardless of query rate. Finally, we note that we also observe similar pattern in our analysis of PSC [13].

Table 1. Histogram of owner service counts at different query rates. Left: uniform distribution; Right: non-uniform distribution

rate\count	1	2	3	4	5	6	7	8	9	11	12	13	15	21	25	28
1	2654	948	133	8	2	0	0	0	0	0	0	0	0	0	0	0
4	10762	3847	472	48	5	3	2	0	0	0	0	0	0	0	0	0
16	42635	15419	1734	177	29	3	5	1	2	1	0	0	1	0	0	0
64	165901	62293	8192	902	138	29	10	0	1	1	2	1	1	2	1	1

rate\count	1	2	3	4	5	6	7	9	10	11	12	19
1	3160	830	64	2	0	0	0	0	0	0	0	0
4	12869	3144	205	16	5	0	0	0	0	0	0	0
16	53232	12203	679	59	7	3	0	0	1	0	0	0
64	216818	47483	2278	160	13	5	1	1	2	1	1	1

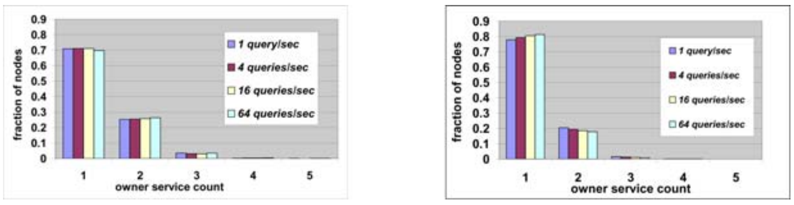

Fig. 7. Fraction of nodes that has various owner service counts. Left: Uniform dist.; Right: Non-Uniform dist

6 Related Works

Tapestry [14] and Pastry [10] are early DHT systems that can locate nearby object. However, since they use randomly hash node id, the "nearby" copy located is in the sense of the node virtual space.More recent efforts of reducing routing stretch and locating nearby object include Coral [3], Canon [4]and BeeHive [8]. Coral divides the P2P network into several levels of self-similar network, each level employs an original chord network. As Coral relies on a mechanism to gradually cluster nodes and split/merge clusters as the network evolves, its effectiveness hinges largely on the efficiency of the cluster algorithm. BeeHive advocates proactive caching in the P2P network to alleviate hot spot problem, relating the amount caching to the query rate. However, in the real world, it may be difficult to decide the query rate a priori, especially in the sudden surge of a flash crowd. The main contribution of this paper is to demonstrate the feasibility of inherently incorporating locality-awareness into P2P overlay. Finally, we note that the hierarchical area structure adopted in Leopard has also been used in other routing/loop-up schemes. For example, GLS [5] uses it in mobile node location lookup. However, as we have shown, P2P look-up service has many unique problems, such as managing multiple copies of an object, constructing a node overlay that preserves neighbor proximity. Leopard is also superficially similar to DIM [6], proposed for sensor networks, with similar terminologies like "zone," "sibling node," etc. In DIM, node neighbor relationships are formed based on geographical closeness (rather than random hashing). However, DIM is mainly designed for range queries, and always stores one copy of an object/event, while Leopard aims to find the nearest copy of an object in the network.

7 Conclusion and Future Work

We have proposed to incorporate locality-awareness inherently into the P2P network and have demonstrated many desirable properties of Leopard. Our future plan includes detailed comparisons between Leopard and Pastry/Tapestry on IP level stretch and capability of finding nearby copy, Leopard operation issues such as dynamically determining level number, and their efficient implementations.

References

1. Global Nework Positioning. http://www-2.cs.cmu.edu/ eugeneng/research/gnp/.
2. F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. of ACM SIGCOMM*, 2004.
3. M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *Proc. of USENIX NDIS*, 2004.
4. P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in g major: Designing dhts with hierarchical structure. In *Proc. of IEEE ICDCS*, 2004.
5. J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proc. of MobiCom*, Aug. 2000.

6. X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 63–75. ACM Press, 2003.
7. T. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. of IEEE INFOCOM*, June 2002.
8. V. Ramasubramanian and E. G. Sirer. Beehive: Design and implementation of next generation name service for the internet. In *Proc. of ACM SIGCOMM*, 2004.
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of ACM SIGCOMM*, Aug. 2001.
10. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale p2p systems. In *Proc. of IFIP/ACM Middleware*, 2001.
11. I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishman. Chord: A scalable P2P lookup protocol for Internet applications. In *Proc. of ACM SIGCOMM*, 2001.
12. L. Tang and M. Crovella. Virtual landmarks for the Internet. In *Proc. of ACM IMC*, Oct. 2003.
13. Y. Yu, S. Lee, and Z.-L. Zhang. Leopard: A locality-aware peer-to-peer system with no hot spot, Tech. Report CSE Dept., U of Minnesota, 2004.
14. B. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1), 2004.