Deetoo: Scalable Unstructured Search Built on a Structured Overlay

Taewoong Choi *Student Member, IEEE,*, Kyoungyong Lee, Renato Figueiredo *Member, IEEE,* and P. Oscar Boykin *Member, IEEE,*

***Abstract*—We present Deetoo, an algorithm to perform completely general queries, for instance high-dimensional proximity queries or regular expression matching, on a P2P network. Deetoo is an efficient unstructured query system on top of existing structured P2P ring topologies. Deetoo provides a reusable search tool to work alongside a DHT, thus, it provides new capabilities while reusing existing P2P models and software. Since our algorithm is for unstructured search, there is no structural relationship between the queries and the network topology and hence no need to provide a mapping of queries onto a fixed DHT structure. Deetoo is optimal in terms of the trade-off in querying and caching cost. For networks of size N, $O(\sqrt{N})$ cost for both caching and querying is required to achieve a constant (in N) search success probability. Queries execute a time of $O(\log^2 N)$. Each of the proposed algorithms are carefully tested and evaluated by simulation, experiments, as well as theoretical proofs. The proposed algotithms are applicable to be built on top of many existing P2P overlays.**

***Index Terms*—Peer-to-peer systems, overlay network, file searching, query flooding, distributed systems**

I. Introduction

SINCE the success of applications like Napster and Gnutella, Peer-to-Peer (P2P) file sharing systems have become some of the most common Internet applications. An Internet study reported that over 73% of all Internet traffic was the result of P2P file-sharing platforms[1]. P2P is the single largest consumer application of bandwidth on networks. P2P traffic significantly outweighs Web traffic and still continues to grow [1].

Each node in a P2P system operates simultaneously as both a server and a client in a distributed fashion. Nodes navigate the underlying network without knowledge of their global structure. Currently, there are two common types of P2P search systems: flooding-based unstructured search systems and Distributed-Hash-Table (DHT)-based key-lookup systems. DHT-based query systems ($O(\log N)$) outperform flooding-based systems in terms of search cost ($O(N)$), but they cannot support general searches because each object and search term must be mapped into the structure of networks. We are interested in building unstructured search systems on top of structured P2P networks. In unstructured search systems, the structure of networks is totally independent of the query matching function, and this enables support of any kind of query.

In this paper we present Deetoo, an efficient query-resolution algorithm based upon Kleinberg's one-dimensional construction [2]. The idea is to organize nodes in a ring in which each node has a set of "local" contacts and one "long-range" contact. A small-world model allows a reduced path discovery with only local information by forming a local tree in the search space. In addition to the search efficiency and low maintenance cost, we can reuse existing P2P code designed for a ring topology by adapting a one-dimensional small-world network model. The heart of our work is simple data object replication and a search algorithm. In our model, the usual 1-D overlay topology is transformed into a rectangular matrix as described in Section III. Replication and query resolution are executed by bounded broadcasting over the columns and rows on the matrix space respectively.

The benefits of Deetoo search algorithm in this work is as follows:

General query: Since there is no need to map data objects or query messages into structured network topology as in DHTs, Deetoo supports any kind of query, such as high-dimensional proximity searches and regular expressions.

Optimal caching and querying cost: Deetoo is optimal search algorithm in terms of trade-off between caching and querying cost to achieve constant query success probability.

Reuse existing P2P design: Deetoo can be built on top of any existing structured P2P topologies, such as Chord, to work alongside a DHT and to minimize developing and deploying P2P software.

Load Balancing: Since replica objects are spread by bounded broadcasting within a randomly selected range, Deetoo achieves uniformly distributed load over entire network.

Efficient data update/deletion: In unstructured search systems, once an item is broadcasted for replication, there is no easy way to trace the location of replicated data. This makes it difficult to update or delete all the replicated data. The range of bounded broadcasting in Deetoo is maintained with each replicated data item. The range information is used for maintaining and for deleting the data items.

This paper evaluates the novel Deetoo algorithms from a theoretical standpoint through simulation to experiment. We focus on the behavior of the search algorithms for each of the following metrics: successful searching probability (hit rate), communication cost (bandwidth consumption or number of generated messages), and search time (depth of the multicasting tree). Measures of the query accuracy are important because precise information retrieval is one of the main purposes of P2P networks. Distributed systems should avoid massive communication cost. Deetoo stores $O(\sqrt{N})$ replicas per object with a high probability. Unlike the structured network, Deetoo

[1]According to http://www.ipoque.com/

does not require that each replicated object be mapped into a DHT. A query generates $O(\sqrt{N})$ messages. This implies more messages when compared to DHTs, which have only O($\log N$) complexity, but offers a major improvement over flooding-based searches with $O(N)$. Assume that data can be cached on C of any N nodes. Since there is no structure assumed, and if load is evenly balanced across the nodes, caching does not depend on the data being cached and we assume that C nodes are selected at random. Similarly, when we query, we can check Q of the N nodes, again at random. Thus the probability we miss the stored data is approximately:

$$\left(1 - \frac{C}{N}\right)^{Q} = \left(1 - \frac{C}{N}\right)^{\frac{N}{C}\frac{CQ}{N}} \approx \exp(-\frac{CQ}{N})$$

So, we need $CQ = O(N)$ in order to have a constant probability of success in this model. This presents an intuitive trade-off, the more nodes on which we cache data, the fewer we need to check to find it. Let p_c be a fraction of nodes that has caching, p_q be a fraction of nodes that has querying, α be a replication factor, and K be a total cost both for the cache and the query. Note that $p_c + p_q = 1$. The replication factor is set to $\alpha = CQ/N$. Then,

$$K = p_c C + p_q Q \quad (1)$$

$$= p_c \frac{\alpha N}{Q} + p_q Q \quad (2)$$

By solving the equation for the minimum K, the minimum cost for query is:

$$Q = \sqrt{\frac{p_c}{p_q}\alpha N} \quad (3)$$

Similarly,

$$C = \sqrt{\frac{p_q}{p_c}\alpha N} \quad (4)$$

To minimize $C+Q$, which is the cost to cache an object and query for it once, the best choice is for C and Q to be $O(\sqrt{N})$. This simple analysis suggests that any unstructured load-balanced system will be require at least this much communication. Of course, logarithmic complexity would be nice, but for many practical systems, the number of nodes might well be 100 to $10,000$ which mean C and Q values on the order of 10 to 100. Such overhead costs would be completely acceptable for many applications, especially when we handle meta data instead of large multimedia files. Deetoo is the first system that arbitrary unstructured queries can be mapped to a structured P2P system with query cost $O(\sqrt{N})$. Our searching algorithm can coexist on the same P2P network with a DHT, so with Deetoo users can still have structured as well as unstructured queries. Also, users can control success rate and search cost with Deetoo by adjusting the degree of replication with a user-specified value. In [3], Lv et al. showed that square-root replication distribution is theoretically optimal in terms of minimizing the overall search traffic. Our protocol search time outperforms other unstructured models by forming an efficient local tree: Deetoo completes searches in $O(\log^2 N)$ time by using Kleinberg's small-world network model. Lastly, due to the fact that Deetoo is built on a structured overlay, objects can be updated or deleted after being published on the system. This represents an advantage over random walk based schemes previously proposed in the literature.

The rest of this paper is organized as follows: First, our searching algorithm is compared with that of the recent routing schemes in the following section. In Section III, the system model used in this work is described. We analyze the performance of Deetoo in Section IV. In Section V, we present simulation results of the performance of Deetoo. The test of Deetoo on a real system is described in Section VI. Finally, conclusions are drawn in Section VIII.

II. Related Works

Search methods for P2P systems can be categorized into two broad systems: DHT-based structured look-ups and unstructured searches using flooding or random walks. Highly structured P2P networks with DHT look-up algorithms [4]–[7] are efficient in that these networks achieve low query costs of $\log N$. This is because they place data objects at particular points on the network topology which are determined by an object's key. Beehive [8] achieves constant look-up latency on top of a DHT through proactive replication. Despite their efficiency, the possibility of operation, even with extremely unreliable nodes, has not been yet examined. In addition, it is impossible for Beehive to handle high-dimensional complex queries. Extensive research has been conducted to address the limits of the exact search problems in DHT. One example is pSearch [9]. A rolling index reduces the number of dimensions for mapping purposes onto the overlay and divides the semantic vector (SV) into sub-vectors. Each sub-vector has the same number of dimensions as the CAN overlay does. Although pSearch is simple and supports high-dimensional queries, it requires control on the data objects of each node. Especially, when nodes join and leave frequently, pSearch incurs massive overhead. Therefore, it is more suitable for networks with stable nodes rather than for highly dynamic networks. pSearch still requires mapping search index into structured P2P overlay, and this limits the support of general query. Cubit [10] provides keyword search capability over a DHT. Cubit efficiently finds multiple closest data sets for a given query. However, it requires the creation of a keyword metric space and only returns multiple similar results to compensate for typos in queries.

More complex query resolution methods have been explored for P2P resource discovery for grids. SWORD [11], Mercury [12], and MAAN [13] are proposed to support multi-attribute range queries on top of structured overlays. In SWORD and MAAN, a DHT is created for each attribute. The number of created DHTs is the same as the number of attributes. Mercury also maintains a logical overlay for each attribute but it does not use DHTs. For few parameters and very narrow range queries, these systems can outperform Deetoo. Otherwise, for large numbers of parameters and larger query ranges, Deetoo can perform better as it always requires the same $O(\sqrt{N})$ complexity regardless of query or data type. Another drawback of DHT-based range query systems is that maintaining multiple overlays costs network traffic because update traffic increases as the number of attributes are increased. Deetoo creates two overlays and update traffic takes place only in the caching overlay. Moreover, because data is not structured, Deetoo's query is not limited to range query. Since the data objects as well as query messages need to be mapped onto network structure in DHT-based systems, there is no easy way for DHT systems to support complex queries such as multi-dimensional queries. Deetoo does not demand any keyword mapping into DHTs and can execute more general queries like regular expression searches.

Unlike DHTs, unstructured P2P systems mostly depend on flooding for message transfers. The big advantage of flooding-based P2P systems is the capacity for high-dimensional search. An early version of Gnutella was based on naive flooding. Because flooding produces a very large number of messages over an entire network, pure flooding limits network size. To address this scaling problem, various types of solutions have been proposed. KaZaa [14] and iXChange [15] introduced central server-like super-peers. However, super-peers cause bottlenecks, security issues, and single point of failure problems due to their server-like characteristics. Recently, research efforts have also focused on locality-based flooding. Systems adopting interest-based locality [16]–[18] assume that two peers having common interests share pieces of a data object. Under this assumption, a shortcut connection is established between two peers having common interests, and queries from one peer are delivered to the other through this shortcut link in the first stage of flooding. Locality-based flooding requires warm-up procedures to gather query history for shortcut connections. LightFlood [19] uses a neighbor-degree-based locality scheme. LightFlood forms a tree-like sub-overlay called FloodNet using neighbors' degree information. Once the sub-overlay is formed, there are two overlay networks in the system: the original P2P overlay and FloodNet. The flooding takes two stages. Messages are transmitted using pure flooding with relatively small TTL values in the first stage. The peers that receive the query with zero TTL trigger the second stage of flooding in the FloodNet. Although LightFlood is simple and helps stop generating massive messages for queries at a certain point, searching unpopular objects requires the entire network to be visited; in addition, LightFlood needs to be warmed up.

A random walk-based search technique is introduced by Adamic et al. in [20]. Their work reduces search cost by the factor of the number of replicas, but they do not consider replica placement. Although random walk searches have an advantage over flooding in terms of search cost, they has some scalability problems because almost all the queries tend to concentrate on the high-degree nodes. To address this problem, object replication using the square-root principle [3], [21] and topology reconstruction [22] have been proposed. Both reduce search time but incur considerable communication cost to maintain fresh topologies or data replication copies. Popularity-biased random walk [23] achieves the square-root principle without the cost of data movement or topology reconstruction. Sarshar et al. [24] combined flooding and random walking in power-law networks. In their work, a query can be resolved in time $O(\log N)$. However, $O(N \times \frac{2logk_{max}}{k_{max}})$ messages are transmitted for a single query, where k_{max} denotes the maximum degree; thus, when $k_{max} = \sqrt{N}$ this becomes $O(\sqrt{N}\log N)$.

Liu et al. [25] studied bounded broadcasting in wireless sensor networks. A balanced push and pull strategy achieves $O(\sqrt{N})$ search cost in the best scenario. However, comb-needle data discovery technique requires nodes to estimate cache and query frequency. All nodes in the network should keep their location information in the grid. Though they do not analyze search time, it is possible to estimate it. By the nature of the hop-by-hop message transfer, the comb-needle data discovery takes linear time in the bounded range which are $O(\sqrt{N})$. The dynamic paths quorum system [26] is scalable and operates in a dynamic setting. The quorum sets are divided into reading quorums and writing quorums in a grid, and each reading quorum intersects each writing quorum. They analyzed probe complexity and availability, especially in a dynamic environment. The probe complexities of the non-adaptive (without stabilization) and adaptive (with stabilization) algorithms is $\Theta(\sqrt{N}\log N)$ and $\Theta(\sqrt{N})$, respectively.

III. The Deetoo Search Algorithm

In this section we describe the data structure and search algorithm we use for Deetoo. We take a similar approach to the idea of DHT P2P networks: take the hash table data structure, and build a distributed data version of this data structure where memory locations spread across many nodes. To understand the Deetoo P2P system, we will first describe a local data structure and then describe a distributed version of that data structure.

A. An unstructured "hash" table

Consider a table data structure that has B bins arranged in a $k \times l$ array ($B = kl$). We can say b_{ij} is the bin in row i column j. To add an object into this table, select a random column and insert the object into each bin in that column. Which is to say, choose a random value $r \in (1, l)$, and insert the object in the set of bins $C_r = \{b_{ir} | i \in (1, k)\}$. To search for an object, select a random row and check each bin in that row for the object. Equivalently, choose a random value $p \in (1, k)$ and look for the object in the set of bins $Q_p = \{b_{pj} | j \in (1, l)\}$. Since every row and column intersect at exactly one bin, $C_r \cap Q_p = \{b_{pr}\}$, a query will always find one bin into which an object was inserted. The number of bin accesses to insert an object is k. The number of bin accesses to query for an object is l. A trade-off between cost of insertion and cost of searching exists.

As a local data structure, the above may have little value: it costs k times more to store than an unsorted list, and the total number of comparisons needed for a search is still M if there are M objects in the table. However, as a distributed data structure designed to distribute load and minimize communication, it is useful since only l bins need to be searched. This data structure achieves totally balanced load distribution in the network because each object is replicated over a bounded region irrespective of its popularity. On current unstructured systems, queries can concentrated on a specific peer having many objects. Besides, popular objects are likely to be available

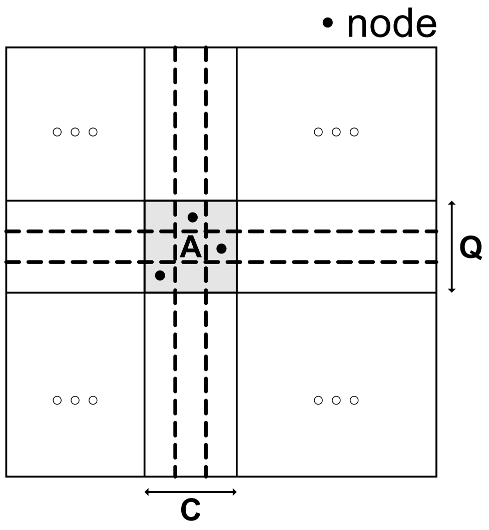


Fig. 1. The caching and querying space.

at several nodes and the probability of succeeding in queries for it is much higher. In other words, queries for rare objects are barely answered. Because our algorithm distributes objects evenly over the network, we can avoid creating *hotspots* and improve probability of finding rare objects. In the next section we describe how to make a distributed version of this data structure which can support general data objects and queries. We will describe a randomized version of the above where queries succeed with a high probability.

B. A Distributed unstructured table

Our setting will be the standard P2P setting: there are N nodes which can communicate, store objects and perform queries. In addition to nodes, there are also bins. Bins may or may not be occupied by a particular node. The number of bins is assumed to be fixed at some very large number, for instance 2^{160}. The bins are arranged in a rectangular array, which in this work we will assume to be square. In the data structure of Section III-A, each row and column intersect at exactly one bin that may or may not be occupied by a node. Instead of querying and inserting along one row and one column, we will do so over a sufficient number of rows and columns such that the probability of having more than one node in the overlapping set is very likely to be one.

Figure 1 depicts the $2-D$ array we use to cache (insert) and query (search). In Figure 1, the area A represents the intersection of a particular cache and query operation. We see in that area, for example, there are three nodes, and so the query will be successful. We need to show several things to see that this algorithm will work in a distributed setting:

1) Show how to efficiently send cache and query messages to entire columns and rows respectively.
2) Show how to deal with nodes joining and leaving the network.

To deal with the above two problems we leverage existing work on building routable $1-D$ P2P networks such as Chord [4] and the small-world model [2]. Rather than build a P2P network that is explicitly two dimensional, we build two sub-graphs, each of which is a 1-D ring, which we can call the query ring and the cache ring. Each P2P node has exactly one address, and hence position, on each ring. This is depicted in Figures 2, 3, and 4; these drawings illustrate how one-dimensional rings for querying and caching locally overlay atop of the 2-D Deetoo array. A node is cached at location i with probability c_i and a query node is placed at location i with probability q_i in the query ring. $N(x)$ and $N(x')$ denote nodes whose addresses are x and x', respectively. x' is transposed address of x.

In order to achieve the effect of ordering the cache ring by increasing along the columns and the query ring to increase along the rows, the bin address for a given node in the query ring must be the transpose of the address on the cache ring, as depicted in Figures 2 and 3. Assume that the size of the address space, and hence number of bins, is $B = m^2$. The address mapping algorithm is as follows: Let x denote an address on the cache ring. The address x can be expressed by column element, x_i, and row element, x_j, such that $x = mx_i + x_j$. A translated address on the query ring x' is obtained by exchanging column element and row element: $x' = mx_j + x_i$.

So, each P2P node then has two addresses in virtual 1-D rings and follows the usual procedure for joining each of the two rings as described in III-E. Notice that on the cache ring, the nodes in the same columns (and adjacent columns) have adjacent addresses. On the query ring, nodes in the same row (and adjacent rows) have adjacent addresses. Because the rings are efficiently routable, it is also efficient to send a message to a randomly selected node near the start of a column or row. Similarly, to reach all elements of a row or column, we can use a bounded broadcast on one of the two rings.

C. Bounded Broadcast

In Deetoo, a bounded broadcast is accomplished with the following recursive algorithm: To broadcast a message over the region $[x, y]$, our routing algorithm finds any node in the given range firstly via greedy routing, in which a node finds the closest node to the destination as its next node. Let us assume that a node x is the first node in the range recognized by greedy routing, then the message is sent to node x. Suppose x has F connections to nodes in the range $[x, y]$. We denote the i^{th} such neighbor as b_i. The node x sends a bounded broadcast over a sub-range, $[b_i, b_{i+1})$, to b_i, except the final neighbor. Differently stated, b_i is in charge of bounded-broadcasting in the sub-range $[b_i, b_{i+1})$. If there is no connection to a node in the sub-range, the recursion is ended and the node stays in the tree as a leaf node. To the final neighbor (b_F), x, sends a bounded broadcast to $[b_F, y]$. When a node receives a message to a range that contains its own address the message is delivered to that node in addition to being routed to others. Figure 5 shows how this bounded broadcast forms a local tree recursively. The time required for this is $O(\log^2 N)$ as shown in Section IV-C. By Deetoo's

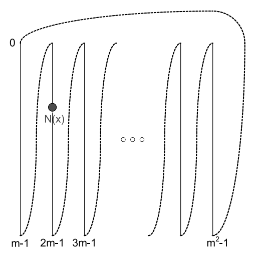


Fig. 2. Virtual ring 1 for caching.

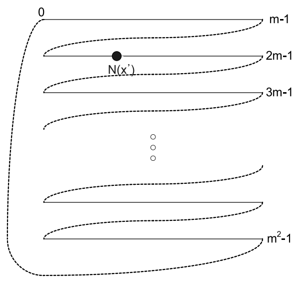


Fig. 3. Virtual ring 2 for querying.

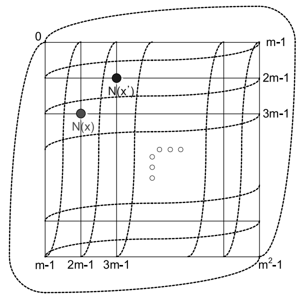


Fig. 4. The complete searching space.

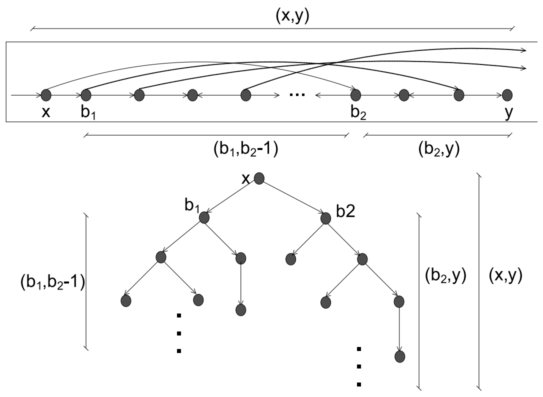


Fig. 5. Bounded Broadcast in range $[x, y]$

recursive bounded broadcasting algorithm, all nodes in the range are involved in forming the local tree because only a node with no connection in the sub-range to the node which is responsible for bounded-broadcasting can stop recursion out of each branch of the tree.

D. Data Insertion and Query

Deetoo is a search algorithm on a specific network structure. When an object is inserted at a node a as in Figure 6, Deetoo selects, at random, a range in which a query message is to be broadcasted locally. Node a starts greedy routing to find a node n in this random range and n starts bounded broadcasting within a limited range on the caching ring to replicate the object (Figure 7). Bounded broadcasting size for caching (C) is given by $C = \alpha\sqrt{\frac{B}{N}}$, where α, a replication factor, is a constant(details about this formulation is followed in Section IV). All nodes in the range of bounded broadcasting eventually receive a copy of an object, o (Figure 8).

Query resolution follows the same bounded broadcasting steps. The only difference is that the query resolution is executed in the querying space in which every node's address is transposed (stated in Section III-B) in the caching space. Bounded broadcasting size for a query, Q, follows the same formulation as the caching size C as shown above. Figures 9, 10, and 11 show the steps involved in a query: node a' issues a query, node n' initiates a bounded broadcast, and node $n(o)$ resolves a query. Since all nodes in the area A have a copy of an object o, n' can retrieve a desired object from any of the nodes in A. By following these steps, a query can be resolved if at least one node exists in A. Note that a user can select bounded broadcasting size by adjusting the *replication factor*, α. The bigger the α, the higher the probability to hit a node in A, at the expense of larger number of messages and replicas.

When an object is obsolete, Deetoo provides a deletion that uses both caching and querying. First, the deleter queries for an object which needs to be deleted. Each object is stored along with the original range into which it was inserted. After the object's range is acquired, the *deletion* message is broadcasted within the range in the same way the object was replicated. As a result, Deetoo guarantees all replicated objects that are supposed to be stored in the nodes in the range to be removed.

E. Node Joins, Leaves, and Stabilization

In this section, we describe how a node joins and leaves the network. Deetoo requires that each data object is cached over the desired range in a network. However, a new node joins the network with no cached objects. Success probability must stay constant regardless of node joins or leaves. Thus, it is necessary that all objects be replicated if a new node's address falls in a object's cached range. Deetoo relies on a new node copying objects from its new neighbors. We define this replicating process as *stabilization* in this paper.

When a new node joins a network, the following steps are taken:

1) select two different random addresses,
2) calculate minimum ring distances to neighboring nodes which have left closest address and right closest address,
3) find a proper place on the virtual ring space by selecting address whose minimum distance to neighbor node is bigger than the other,
4) obtain an address and makes a connection to two adjacent neighbors and one short-cut neighbor based on Kleinberg's *inverse* r^{th}*-power distribution*,

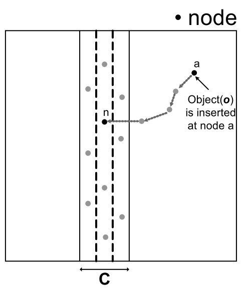


Fig. 6. An object(o) is inserted at node a. The message is routed to a node(n) in the region for the bounded broadcast.

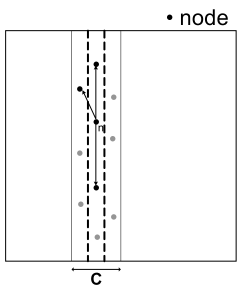


Fig. 7. The bounded broadcast starts at node n.

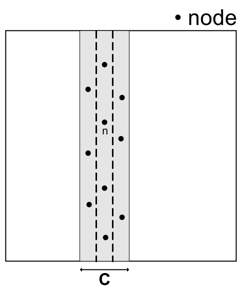


Fig. 8. All nodes in the range receive the message.

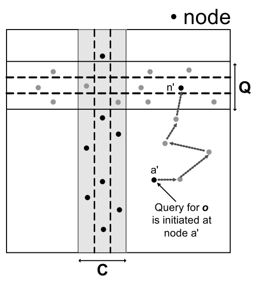


Fig. 9. A query for object o is initiated by a node a'.

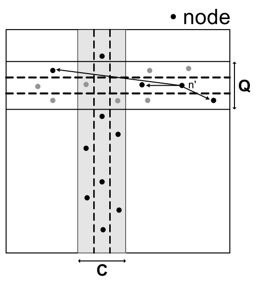


Fig. 10. n' starts bounded broadcasting within Q

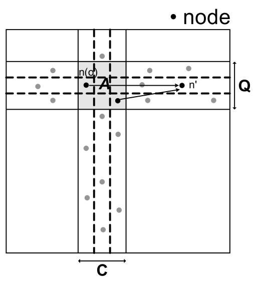


Fig. 11. Finally, n' retrieves an object from a node $n(o)$.

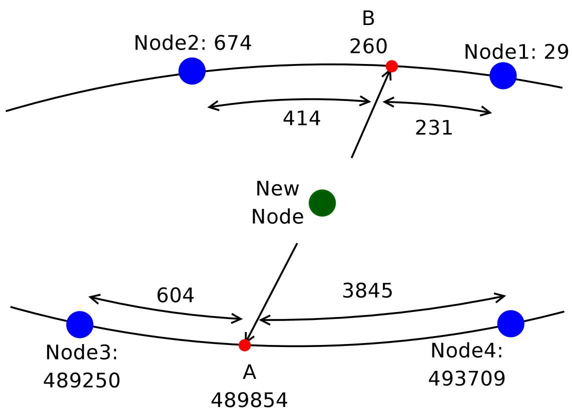


Fig. 12. Choosing a position on the ring when a node joins Deetoo.

5) as a final step, copy objects from neighbors in the same set of sub-rings (in the same columns in matrix space).

Figure 12 shows how a new node finds its right place. From randomly selected positions A and B, the nearest neighbors are *node1* and *node3*. Between these two nodes, *node3* has a bigger distance to A than the distance from *node1* to B. Thus, a new node takes a position A as its new address.

Each object must keep its range information as well as the object itself or meta data for the object where it was initially inserted to the network so that as nodes join and leave the objects can be maintained on their randomly selected column. After a node obtains a proper address successfully, the node sends requests to its newly connected neighbors in order to replicate objects in the network. *Stabilization* plays a role in this stage of replication. The following describes how all objects in the same boundary are copied to the new successfully joined node from neighbors.

1) Retrieve the object's range from the neighbor's object list.
2) Recalculate range with a given replication factor.
3) Copy the object from the neighbor node to the newly joined node only if the new node's address lies on the recalculated range.
4) Repeat 1 through 3 for all objects from both adjacent neighbors.

If we assume that there exist m unique objects in the network whose size is N, each node maintains $m\frac{\alpha}{\sqrt{N}}$ objects on average, where α is a replication factor. In the stage of stabilization, the node is able to receive objects from both neighbors. The maximum number of objects which need to be transferred is limited to $O(\frac{1}{\sqrt{N}})$. We analyze *stabilization cost* in Section IV-D. We also simulate stabilization cost at the time of new node joins under churn and compare the simulation result with an analytical model. The simulation result shows that Deetoo requires very low stabilization cost. When a node leaves or fails, our protocol does not need to do anything because all objects have already been copied to all nodes in the same set of columns. Thus, neighbor nodes keep exactly the copies of the objects held by leaving or failing nodes.

F. Estimating Network Size

Each node in a Deetoo network is distributed without any server-client interaction, which makes it impossible for the peers to know exactly how many other peers exist. However, each node is required to know the network size in certain processes. For example, we cannot decide a caching or a querying size without knowing the number of peers in the network. The accuracy of network size estimation is key to deciding the optimal caching and querying range. Both underestimation and overestimation affect performance in success probability and communication cost. If a node underestimates the size of the network, the success probability goes down below average while the communication cost is reduced. The exact opposite result is expected if a bigger size is estimated.

Deetoo introduces two methods to reduce network size estimation error. The first method is that Deetoo helps a newly joining node finding a right place. In the previous section, we explained how a node selects a new address between two candidates. Although choosing a random address is much simpler than our method, our method makes distances between every two nodes more evenly distributed because it always maximizes the minimum distance between two adjacent nodes. As a result, the variance of distance distribution between two nodes is reduced. Next, the average distance between nodes is estimated using a multi-hop-away node. To utilize this approach, we divided this process into two steps. First, the distance between two direct neighbors is calculated to get the initial guess for a network size, N_0. Let an address have b bits; then the whole address space is B=2^b, which is equal to ring distance D_{ring}. Node n calculates the average distance to its neighbors (D_0). Then, the first network size, $N_0(\simeq \frac{B}{D_0})$ is estimated. Second, the average distance, D_1, from n to a node which is $\log N_0$ hops away from n is counted. The final network size estimated, $N_{est}(\simeq \frac{B}{D_1})$ can be evaluated. We experimented under various settings to get the best estimation of the network size. If more nodes are involved in calculating the average distance between two nodes, the level of accuracy goes up though it requires more time and bandwidth. On the contrary, the variance of the average distance is much higher when only two direct neighbors are used for the calculation. We lose accuracy of size estimation in this case though we save time and bandwidth. We observed that our calculation of the average distance using $\log N$ hops is acceptable considering both accuracy and cost. Figure 13 shows estimated network size distribution under our network size estimating algorithm. In Figure 19, the simulation results show that our network size estimation works well in that the success probability is very close to the theoretical expectation.

IV. Analysis of Deetoo

Our analysis focuses on the probability of a query being successfully resolved , P_s, and the communication cost, K, to show that the Deetoo protocol for unstructured P2P networks is efficient and scalable. We build the following assumptions to simplify our analytical modeling. First, we have a fixed size of the addresses. Thus address space, B, is fixed, which is $m \times m$ for cache and query respectively as shown in Figure 1. Second, we assume that the caching probability we cache at location i, c_i, and the querying probability we put a query at location j, q_j, are both uniformly distributed. Third, we assume that the caching and querying probability are the same at any location. Finally, at most one node per a bin in matrix space is allowed for all nodes to have a unique address.

A. Success Probability

In Deetoo, queries are unstructured and load-balanced, which means that the probability that a given node receives a query is independent of the query and equal for all nodes. For each object and query pair, there is a region of size A in the grid address space that receives both the query and the cache. The only way the query will not be resolved is that the region has no node. Since every configuration of N nodes in the address space of size B is equally likely, the probability there is a miss is calculated as following: First, I provide an exact analysis for miss probability. Assume that there are N bins with C being occupied, and I search Q to try to find an occupied bin. The probability I fail to search is calculated as following:

- All C occupied bins must be disjoint from the Q I check and the number of possible combination is $\binom{N-Q}{C}$. Note that if $Q + C \geq N$, $\binom{N-Q}{C} = 0$
- The number of ways to arrange occupied bins is $\binom{N}{C}$

Thus, the probability of miss hit is:

$$P_{miss} = \frac{\binom{N-Q}{C}}{\binom{N}{C}} \quad (5)$$

$$= \frac{(N-Q)!}{N!}\frac{(N-C)!}{(N-C-Q)!} \quad (6)$$

There is four possible cases for the combination of the size of C and Q. I analyze the upper bound and the lower bound for each case.

- case 0: If $Q = 0$ and $C = 0$, $P_{miss} = 1$.
- case 1: If $Q + C > N$, $P_{miss} = 0$.
- case 2: If $Q + C = N$, $P_{miss} = \frac{1}{\binom{N}{C}}$. since $N \leq \binom{N}{C} \leq 2^{-N}$ for $C \geq 1$,

$$2^{-N} \leq P_{miss} \leq \frac{1}{N} \quad (7)$$

- case 3: Q+C+E=N, where $E \geq 1$. In this case, at least one bin is not occupied or searched. I use Sterling's bounds to analyze the bounds for this case:

$$e^{-\frac{1}{12n+1}} < \frac{n!}{\sqrt{2\pi n}n^n e^{-n}} < e^{-\frac{1}{12n}} \quad (8)$$

The cases are fairly trivial to evaluate except the sparse case (case 3) , which is to say the query/cache overlap region and the number of nodes are both small compared to the total

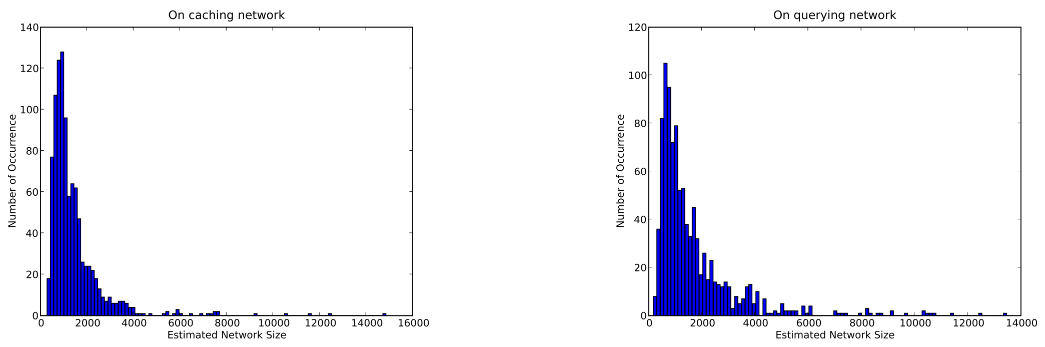


Fig. 13. Estimation size distribution. The actual network size is 1,000. The averages are 1430.97 and 1695.95 for caching and querying, respectively. The standard deviations are 1248.56 and 1630.41, respectively. The medians are 1077 and 1144, respectively. Note that the error margin of overestimation is much larger than that of underestimation, which causes the average to be slightly larger than the actual size of network. Network size estimation for cache is more accurate than that for query, because new node's address selection is based on caching address.

number of bins E can be thought of as the number of empty bins in the case of a miss. Look at the upper bound first.

$$\begin{aligned}
P_{miss} &< \frac{\sqrt{N-Q}(N-Q)^{N-Q}e^{-(N-Q)}e^{-\frac{1}{12(N-Q)}}\sqrt{N-C}(N-C)^{N-C}e^{-(N-C)}e^{-\frac{1}{12(N-C)}}}{\sqrt{N}N^Ne^{-N}e^{-\frac{1}{12N+1}}\sqrt{N-C-Q}(N-C-Q)^{N-C-Q}e^{-N+C+Q}e^{-\frac{1}{12(N-C-Q)+1}}} \\
&= \sqrt{\frac{(N-Q)(N-C)}{N(N-C-Q)}}\left(\frac{(N-Q)(N-C)}{N(N-C-Q)}\right)^N\left(\frac{N-C-Q}{N-Q}\right)^Q\left(\frac{N-C-Q}{N-C}\right)^C e^\gamma \\
&= \left(\frac{(N-Q)(N-C)}{N(N-C-Q)}\right)^{N+\frac{1}{2}}\left(\frac{N-C-Q}{N-Q}\right)^Q\left(\frac{N-C-Q}{N-C}\right)^C e^\gamma \\
&= \left(\frac{(N-Q)(N-C)}{NE}\right)^{E+Q+C+\frac{1}{2}}\left(\frac{E}{N-Q}\right)^Q\left(\frac{E}{N-C}\right)^C e^\gamma \\
&= \left(\frac{(N-Q)(N-C)}{NE}\right)^{E+\frac{1}{2}}\left(\frac{N-C}{N}\right)^Q\left(\frac{N-Q}{N}\right)^C e^\gamma
\end{aligned}$$

Due to the convexity of $\log(1 \pm x)$, we know that

$$(1-x)^\sigma < e^{-\sigma x} \tag{9}$$

$$(1+x)^\sigma < e^{\sigma x} \tag{10}$$

Thus,

$$\left(\frac{N-C}{N}\right)^Q = \left(1-\frac{C}{N}\right)^Q < e^{-CQ/N} \tag{11}$$

$$\left(\frac{N-Q}{N}\right)^C = \left(1-\frac{Q}{N}\right)^C < e^{-CQ/N} \tag{12}$$

$$\left(1+\frac{QC}{NE}\right)^E < e^{CQ/N} \tag{13}$$

Finally,

$$P_{miss} < \left(1+\frac{QC}{NE}\right)^{E+\frac{1}{2}} e^{-\frac{2QC}{N}}e^\gamma \tag{14}$$

$$< e^{-\frac{QC}{N}}\sqrt{1+\frac{QC}{NE}}e^\gamma \tag{15}$$

where

$$e^\gamma = e^{-\frac{1}{12(N-Q)}-\frac{1}{12(N-C)}+\frac{1}{12N+1}+\frac{1}{12(N-C-Q)+1}} \tag{16}$$

$$< e^{\frac{1}{12E+1}} \tag{17}$$

Thus if I set C and Q such that $CQ = \alpha B/N$, I have $P_{miss} < e^{-\alpha}\sqrt{1+\alpha/E}e^{\frac{1}{12E+1}}$.

One can find a similar lower bound:

$$\begin{aligned}
P_{miss} &= \frac{\sqrt{N-Q}(N-Q)^{N-Q}e^{-(N-Q)}e^{-\frac{1}{12(N-Q)}}\sqrt{N-C}(N-C)}{\sqrt{N}N^Ne^{-N}e^{-\frac{1}{12N}}\sqrt{N-C-Q}(N-C-Q)^{N-C-Q}} \\
&= \sqrt{\frac{(N-Q)(N-C)}{N(N-C-Q)}}\left(\frac{(N-Q)(N-C)}{N(N-C-Q)}\right)^N\left(\frac{N-C-Q}{N-Q}\right. \\
&= \left(\frac{(N-Q)(N-C)}{N(N-C-Q)}\right)^{N+\frac{1}{2}}\left(\frac{N-C-Q}{N-Q}\right)^Q\left(\frac{N-C-Q}{N-C}\right. \\
&= \left(\frac{(N-Q)(N-C)}{NE}\right)^{E+Q+C+\frac{1}{2}}\left(\frac{E}{N-Q}\right)^Q\left(\frac{E}{N-C}\right)^C e \\
&= \left(\frac{(N-Q)(N-C)}{NE}\right)^{E+\frac{1}{2}}\left(\frac{N-C}{N}\right)^Q\left(\frac{N-Q}{N}\right)^C e^\delta \\
&= \left(1+\frac{QC}{NE}\right)^E\left(1-\frac{C}{N}\right)^Q\left(1-\frac{Q}{N}\right)^C e^\delta\sqrt{1+\frac{QC}{NE}}
\end{aligned}$$

where

$$e^\delta > e^{-\frac{1}{12(N-Q)+1}}e^{-\frac{1}{12(N-C)+1}}e^{-\frac{1}{12N}}e^{-\frac{1}{12E}} \tag{18}$$

$$> e^{-\frac{1}{12(E+Q)+1}} \tag{19}$$

$$> e^{-\frac{1}{12E}} \tag{20}$$

since $12E < 12(E+C)+1$ and $e^{-\frac{1}{12(E+C)+1}+\frac{1}{12E}} > 1$. We know that the following inequality with some positive real number σ,

$$(1+x)^\sigma > 1+\sigma x \tag{21}$$

So,

$$\left(1+\frac{QC}{NE}\right)^E > 1+\frac{QC}{N} \tag{22}$$

Thus, the lower bound is obtained:

$$P_{miss} > \left(1+\frac{QC}{N}\right)\left(1-\frac{C}{N}\right)^Q\left(1-\frac{Q}{N}\right)^C\sqrt{1+\frac{QC}{NE}}e^{-\frac{1}{12E}} \tag{23}$$

$$> (1+\alpha)\left(1-\frac{\alpha}{Q}\right)^Q\left(1-\frac{\alpha}{C}\right)^C\sqrt{1+\frac{\alpha}{E}}e^{-\frac{1}{12E}} \tag{24}$$

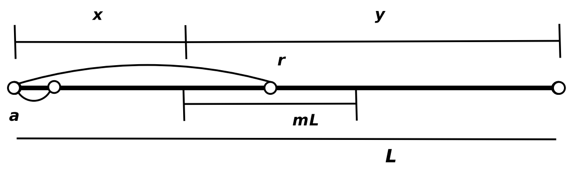


Fig. 14. The probability that a node *a* has a connection to a node *r* in the middle(region *mL*) is defined as p_m, and p_m is inverse proportional to the distance between *a* and *r*.

The lower bound shows that $P_{miss} \to 1$ as $\alpha \to 0$. If CQ/N goes to zero as N gets large, $P_{miss} \to 1$, and this result can be extended to any system with independent queries and caching and where each node is equally likely to be used for each query or cache, hence Deetoo is optimal in terms of the asymptotic of query/cache cost trade-off. When E is large, as is the case for large networks with large address spaces and non-negligible caching, $P_{miss} \approx e^{-\alpha}$. So, we are able to have success probability solely as a function of the constant replication factor α, and independent of the number of the nodes.

B. Communication cost

In order to check the network's scalability, communication costs need to be examined. Each caching message is transferred to all the nodes in the shaded area in Figure 8, and this area has $C\sqrt{B}$ bins. Since we assume that cache and query probabilities are equal, $C = Q = \sqrt{\alpha B/N}$; therefore either caching cost or querying cost K is:

$$\begin{aligned} K &= pC\sqrt{B} \text{ (or } pQ\sqrt{B}) \\ &= \frac{N}{B}C\sqrt{B} = \frac{N}{B}\sqrt{\frac{\alpha B}{N}}\sqrt{B} \\ &= \sqrt{\alpha N} \end{aligned}$$

From the above analysis, we show that a query in the network generates $O(\sqrt{N})$ messages.

C. Search time

Search time can be analyzed by computing the average depth of the tree formed by the bounded broadcast presented in Section III-C. When we assume that each node has a connection to its nearest neighbors on both rings as well as one shortcut connection to a node at distance d with probability proportional to $1/d$, we can show that the expected depth of the tree is $O(\log^2 N)$. Our proof is analogous to computing the average time to reach a node using greedy routing as in [7], [27], while search time of Deetoo measures the average maximum depth of the local trees. From the Kleinberg's model, each node has a long-range contact at distance *r* with probability $p_r = \frac{1}{\log N}\frac{1}{r}$ as in Figure 14. Let p_m be the probability that node *a* has a long-range contact in the middle, mL region.

$$\begin{aligned} p_m &= \sum_{r=\frac{1-m}{2}L}^{\frac{1+m}{2}L} p_r \\ &= \sum_{r=\frac{1-m}{2}L}^{\frac{1+m}{2}L} \frac{1}{\log N}\frac{1}{r} \\ &\simeq \frac{1}{\log N}\log\left(\frac{1+m}{1-m}\right) \end{aligned}$$

Since mL is a subset of L, $m \in (0,1)$. Let $T(i)$ be the search time in region of size i. Each long range connection either goes to a node in the middle region or not. If we make it into the middle region, the remaining area to search is at most the length of $\frac{1}{2}L + \frac{m}{2}L$. Thus, if the long-range connection goes to the middle, the time is $T(L) \leq 1 + T((\frac{1+m}{2})L)$. Otherwise, we go to the next neighbor so $T(L) = 1 + T(L-1)$. We know that $T(L-1) < T(L)$. If p_m is the probability of making it into the middle, on average, we need to try $1/p_m$ neighbors before we are likely to find a connection to the middle. Putting this together, the average search time is $T(L) \leq \frac{1}{p_m} + T((\frac{1+m}{2})L)$. The first part of the right side of the equation represents time to reach a connection in the middle (*x* in Figure 14) and the second part is time for the rest of the region (*y* in Figure 14. For the rest of the region, at most γ more steps are required to cover the whole region *L*. We know $T(\log N) \leq \log N$. Solving for γ such that $(\frac{1+m}{2})^{\gamma} L = \log N$,

$$\gamma \leq \frac{\frac{1}{2}\log(\alpha N)}{\log(\frac{1+m}{2})}$$

The maximum depth is decided to γ steps multiplied by the number of the nodes to reach a connection in the middle.

$$\begin{aligned} T(L) &\leq \frac{1}{p_m}\gamma \\ &= \frac{\log N}{\log(\frac{1+m}{1-m})}\frac{\frac{1}{2}\log(\alpha N)}{\log(\frac{1+m}{2})} \\ &\leq \frac{\frac{1}{2}(\log^2(\alpha N) - \log\alpha\log(\alpha N))}{\log(\frac{1+m}{1-m})\log(\frac{1+m}{2})} \end{aligned}$$

We calculate an upper bound with the above inequality by getting the maximum of the denominator. At a value of $m \approx 0.517$, the denominator of the right side of the above inequality has a minimum. In consequence, the upper bound for $T(L)$ is minimized. We verify this claim using simulations which are presented in Figure 18. For the comparison, the minimum of upper bound, where *m* is set to 0.5, is calculated and compared with our simulation result.

D. Stabilization Cost

We regard the number of objects that need to be copied to a newly joined node as the stabilization cost because there is no other major message transmission except this copying for a leaving or joining node. We assume that our algorithm checks whether an object is already replicated from either neighbor or not. If a node identifies the object, it discontinues the object transmission. This ensures that a node prevents unnecessary message generation and a network saves bandwidth. Objects which already exist in a new node as well as out of object's range are excluded from transmission. A new node examines each object's range whenever it is ready to be copied. In P2P networks, network size often varies. Thus, as the network size changes, the object's range is recalculated. Caching in a redefined range constrains the number of cached replicas to remain $O(\sqrt{N})$.

Let C_s represent the stabilization cost, k the number of unique objects in the network, N_1 the left neighbor, and N_2 the right neighbor. Let η_i symbolize a probability of the object in node N_i. η_{ij} is a probability of the object in both N_i and N_j and $\eta_{j|i}$ denotes a probability of the object in N_j given that the object is in N_i.

$$\begin{aligned} C_s &= kP(object\ in\ N_1\ or\ N_2) \\ &= k(P(object\ in\ N_1) + P(object\ in\ N_2) \\ &\quad -P(object\ in\ both\ N_1\ and\ N_2) \\ &= k(\eta_1 + \eta_2 - \eta_{12}) \\ &= k(\eta_1 + \eta_2 - \eta_1\eta_{2|1}) \end{aligned}$$

The probabilities, η_i and η_{ij}, are calculated as follows:

$$\begin{aligned} \eta_1 &= \eta_2 = \frac{L}{B} \\ &= \frac{\sqrt{\alpha N} d_{ave}}{d_{ave} N} \\ &= \sqrt{\frac{\alpha}{N}} \\ \eta_{12} &= \eta_1\eta_{2|1} \\ &= \sqrt{\frac{\alpha}{N}}\left(1 - \frac{d_{ave}}{L}\right) \\ &= \sqrt{\frac{\alpha}{N}}\left(1 - \frac{1}{\sqrt{\alpha N}}\right) \end{aligned}$$

Thus,

$$\begin{aligned} C_s &= k\left(2\sqrt{\frac{\alpha}{N}} - \sqrt{\frac{\alpha}{N}}\left(1 - \frac{1}{\sqrt{\alpha N}}\right)\right) \\ &= k(\sqrt{\frac{\alpha}{N}} + \frac{1}{N}) \\ &= O\left(k\sqrt{\frac{\alpha}{N}}\right) \end{aligned}$$

where d_{ave} denotes the average distance between two nodes, N, B, and L represent the number of nodes, the ring distance (the average distance times total number of nodes), and the length of bounded broadcasting range (the average distance times the number of nodes in the range), respectively.

V. Simulation Results

In this section, we confirm the theoretical results discussed in the previous section via simulations. We used Netmodeler[2] for simulating the Deetoo algorithm. In our topology model on top of Netmodeler, each bin may be occupied by at most one node and the nodes form two rings, one for the cache and the other for query. Then, each node is connected to a long-range neighbor according to the *inverse r^{th}-power distribution* for the small-world network model. For simplicity, the size of the query range is assumed to be the same as the size of the cache range. All caching and querying processes are initiated on randomly selected nodes and messages are broadcasted within randomly chosen ranges.

[2]Netmodeler is a free software/open source C++ library developed by the authors

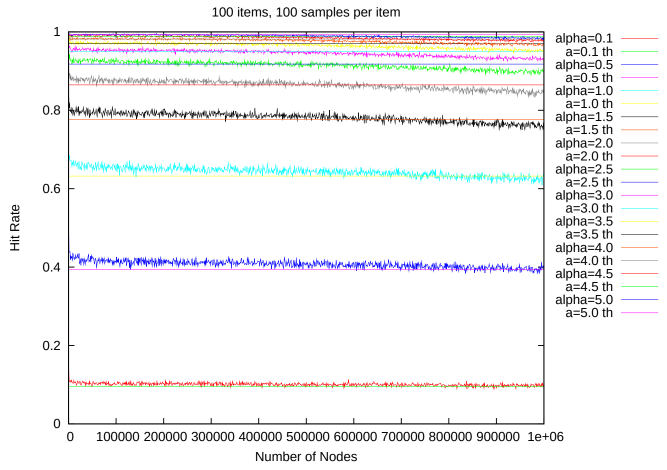


Fig. 15. Query hit rate is constant in network size N, and scales as $1-e^{-\alpha}$.In the legend, "th" is theory, the other is simulated.

A. *Performance Evaluation*

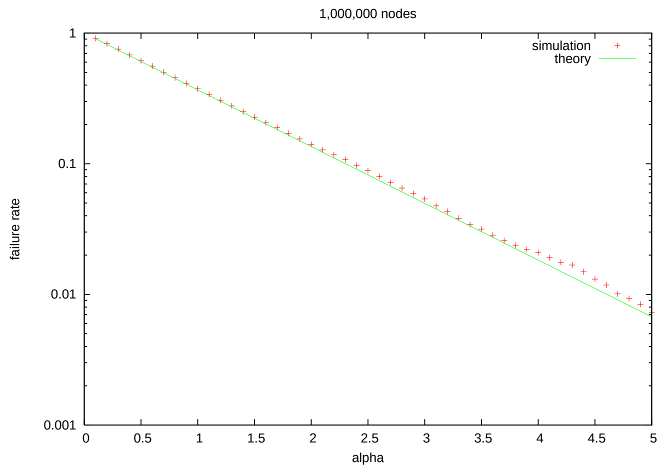


Fig. 16. Query miss rate as a function of α, this scales as $\exp(-\alpha)$.

In our simulations the size of the address space is set to 32 bits which has 2^{32} bins. We count the number of hops as an indication of communication cost and assume that the per-hop communication costs for cache and query are the same. Our simulations are performed on networks of size 10 to 10^6. The number of columns, C, is chosen such that C is equal to the number of rows, Q, $C^2 = \alpha N$. Thus, we simulate various values of α to observe the effect of increasing or decreasing the cache or query range size (α=$\frac{C^2}{N}$). We performed simulations with α=0.1 to 5.0, with intermediate steps of size 0.1. 100 string objects were generated by using a

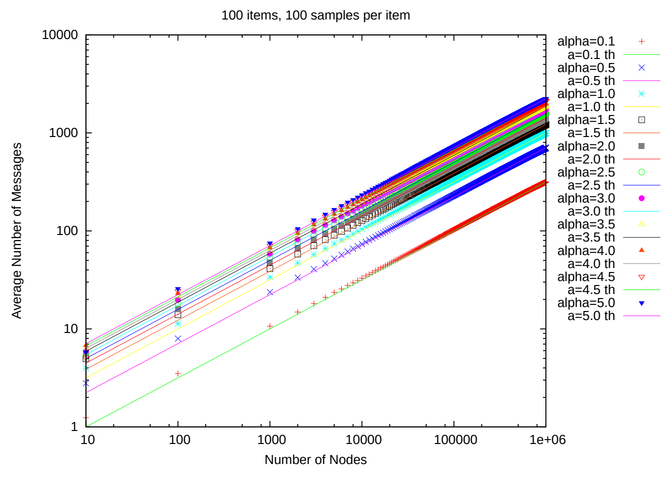


Fig. 17. Query cost scales $O(\sqrt{N})$. "th" is for the theoretical result, and the other is the simulation result.

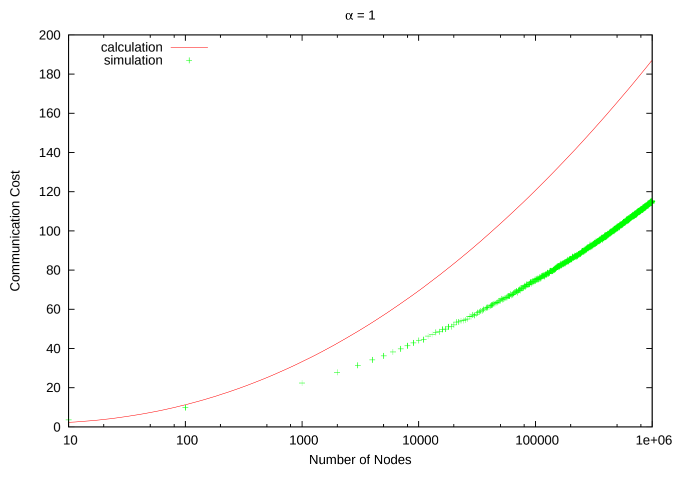


Fig. 18. Search time: We see our upper bound is loose, but the search time grows more than $\log N$, less than $\log^2 N$.

random string generator. Each object was initially inserted on a uniformly randomly selected node. For queries, we count only exact matching objects even though Deetoo can perform partial matches. Each test was repeated 100 times and the results were averaged. Figure 15 shows that query success probability is constant regardless of the network size. The constant success rate is desirable since Deetoo can perform the search with preferred success probability by adjusting broadcasting range with different α. In the figure, we also compare the simulation results with the theoretical results. We observe that the larger the α is, the higher the success probability is. In Figure 16, query success probability declines exponentially as expected. Figure 17 shows communication cost with respect to network size. There is a trade-off between success probability and the communication cost.

For the search time, we measured the number of out-of-range links before finding a node in the range (by greedy routing) and the depth of the multicasting tree. Message transmission times at each link are assumed to be all the same. Figure 18 compares simulation results with calculations with α=1. As mentioned in Section IV-C, 0.5 is substituted for m. Note that *x* axis scales logarithmically. Our simulation shows a loose upper bound for search time, but it is obvious that the scaling of simulation result is more than $\log N$.

B. Numerical Analysis for Success Probability

As described in Section IV-A, the success probability remains constant regardless of the number of nodes in Deetoo network. However, the accuracy of network size estimation affects success probability. The simulation results show slight trend downward of success probability as network size grows. The main reason for the slight decrease of success probability is that network growth makes variance of size estimation increase. In a bigger network, there are higher chances for a node to underestimate its network size relative to a small network. Underestimation is much more harmful for success probability because underestimation of the network size results in a narrower caching or querying range for bounded broadcasting. With the estimated network size distribution, we analyzed the impact of estimation on success probability numerically. We define w_c and w_q as the width of the the columns for caching and querying respectively. Since different nodes will initiate a cache and a matching query, we can consider what happens when the two nodes have difference estimates of the network size.

$$w_c\sqrt{B} = \sqrt{\alpha N}$$
$$w_c = \sqrt{\frac{\alpha N}{B}}$$

Similarly, $w_q = \sqrt{\frac{\alpha N}{B}}$. Thus, the number of bins in overlap is:

$$w_c w_q = \alpha\frac{\sqrt{N_1 N_2}}{B}$$

Assume that N_1 and N_2 are network size estimation for cache and query, respectively. The success probability given a network size N is:

$$\begin{aligned} P_{success}|N &= \left(1-\frac{N}{B}\right)^{\alpha\frac{\sqrt{N_1N_2}}{B}} \\ &= \left(1-\frac{N}{B}\right)^{\frac{N}{B}(\alpha\frac{\sqrt{N_1N_2}}{N})} \\ &\approx 1-e^{\alpha\frac{\sqrt{N_1N_2}}{N}} \end{aligned}$$

Finally, we calculate the success probability with given network size and estimated distribution for both cache and query. To do this, we need to know the probability of estimating the size of the network given the actual size, $P(N_1|N)$:

$$\langle P_{success}|N\rangle = \sum_{N_1=0}^{\infty}\sum_{N_2=0}^{\infty} P(N_1|N)P(N_2|N)\left(1-e^{\alpha\frac{\sqrt{N_1N_2}}{N}}\right)$$

Using our simulation data to obtain $P(N_i|N)$, figure 19 shows that both the numerically evaluated success probability and the simulation results for success probability decrease slightly as network size grows compared to the theoretical

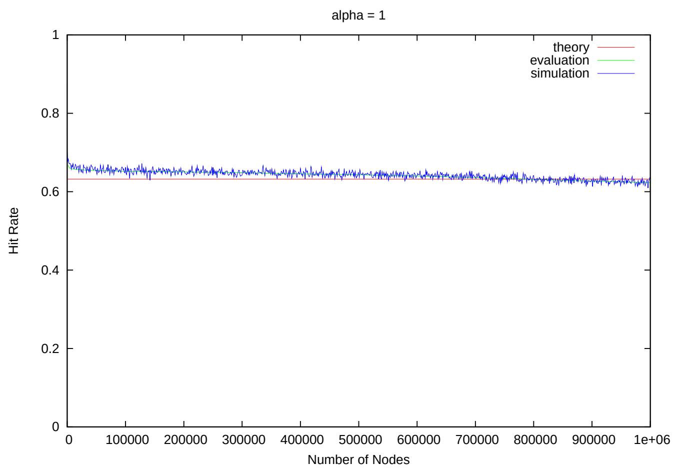


Fig. 19. Success Rate decreases slightly as network size grows, while expected success rate stays constant. Estimation of the network size has bigger error margin in bigger networks. The simulation result depends on the estimation distribution.

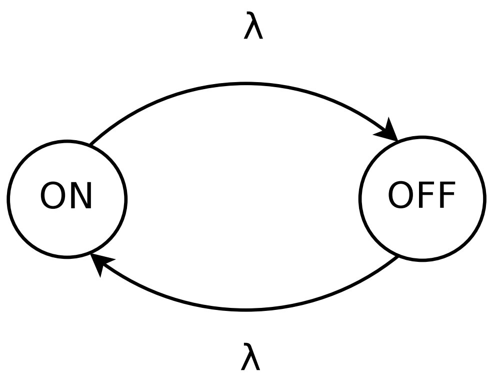


Fig. 20. State Transition Diagram for each node. Because join time and leave time distribution has same average, the probability that each node is alive at a time t is 0.5.

result, which assumes a global knowledge of the network size. Even though there is some noise in the simulation results, we see that the simulation result matches the numerical evaluation fairly well. Mean square error between numerical evaluation and simulation result is 3.93e-05.

C. Robustness

Due to the nature of P2P networks, each peer can leave or fail at any time. Therefore it is important to analyze the effect of node failures. We will show that data objects are still accessible without generating excessive managing overheads under dynamic networks. We extended Netmodelerto model massive churn on nodes. The simulator setting is described below. We set the network size first. Each node joins the network sequentially following a uniform distribution. Upon completing network formation with a given size, each node repeatedly leaves and joins. A node's rejoin time and leave time is exponentially distributed with same mean. The consequence of this distribution makes the average number of alive nodes in the system remains half of total number of the nodes originally joined. As seen in the Figure 20, a node's state transits between *ON* and *OFF* with probability of 0.5. In other words, the probability that each node is active at time t is $\frac{1}{2}$. In our simulation setting, every node stays to be turned *ON* until network size reaches a given number. It takes time a network size is saturated at the half of initial network size on average. After all nodes joined network, 100 data objects were inserted. Queries were executed 100 times per data object. Both the cache events and the query events occur in a time distributed exponentially. For simplicity, a replication factor, α, is set to 1 for the following simulations under churn.

Figure 21 demonstrates how the churn affects success probability. Note that success probability remains constant. Also, the success probability still follows the theory as described in Section IV-A. This result shows that Deetoo is robust against network dynamics.

Under the heavy churn in the network, message transfers for maintenance purposes are not negligible. When a new

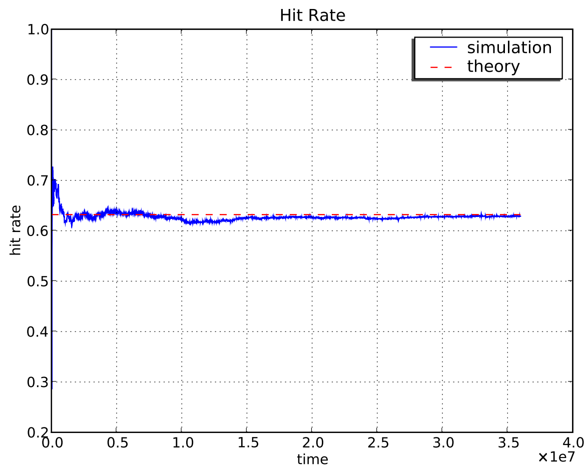


Fig. 21. Success probability remains constant under Churn

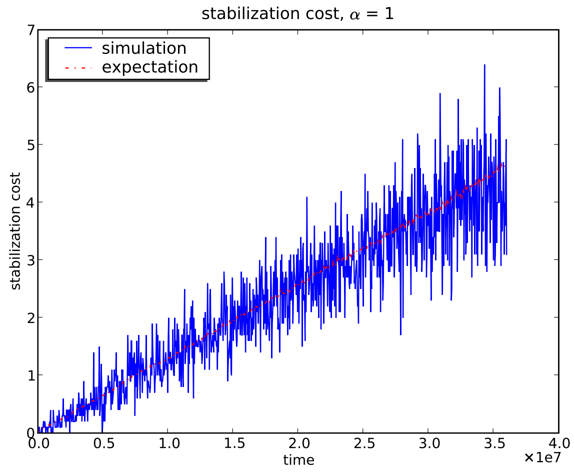


Fig. 22. Stabilization Cost

node joins the network, it is responsible for maintaining data objects if its address is within objects' range. We counted what fraction of the total objects were transferred to a new node. Assume that there are k objects, and each are replicated over $\sqrt{N}$ nodes, we have $k\ \sqrt{N}$ total objects in the system. These objects are placed on nodes in the range selected uniformly at random, so the expected number of objects on each node is $k/\sqrt{N}$. Thus, joining cost is directly proportional to $1/\sqrt{N}$.

Figure 22 shows that how many replication occurred per unique data object after joining new nodes. For the measure of stabilization cost, the size of network is set to 1,000 nodes initially. Note that stabilization cost grows not as time continues, but as the number of unique objects increases. Unlike DHT-based stabilization, our process is much simpler, and costs less.

With very low maintenance cost, Deetoo is capable of efficient data retrieval even under heavy churn.

VI. Deetoo Test on Planet-Lab

In this section, Deetoo is tested in real P2P environment and compared to theoretical expectation as well as simulation results to verify the performance of Deetoo. We built Deetoo as service on top of Brunet [28] and deployed Deetoo-enabled nodes in Planet-Lab [29].

Brunet is a library for P2P networking, which provides the core services of routing, object storage/lookup, and overlay connection management supporting multiple transports (TCP, UDP, and tunnels) and NAT traversal. Many different types of services such as IPOP [30], WOW [31], grid-appliance [32], and socialVPNs [33] are already built on Brunet and some of them provide their service as open software. While Deetoo can be built on top of many P2P applications, Brunet is used for the underlying P2P substrate for the P2P network management in this work. Deetoo‘s computational component for caching and querying is implemented using MapReduce approach in this test. To handle large scale of data-intensive computation in distributed systems effectively, parallel processing in data partitioning and computation is a good way to implement to boost up the speed of data processing. MapReduce concept comes right from supporting such parallel computation. In Deetoo, Map is defined as object matching function for querying network and as object replicating function for caching network. Reduce collects Map results both from children nodes and from the node. Reduce also counts the number of nodes and the depth of the down tree to keep track of network resource usage.

A. System Model

While Deetoo is applicable to many ring-based P2P overlays, Deetoo's functionality is built on top of Brunet P2P overlay networks. The use of Brunet overlay makes it easy to deal with many difficult issues related to network management including the management of connections and NAT traversals. The network topology of Brunet and how Brunet manages connections and routing are described in the section.

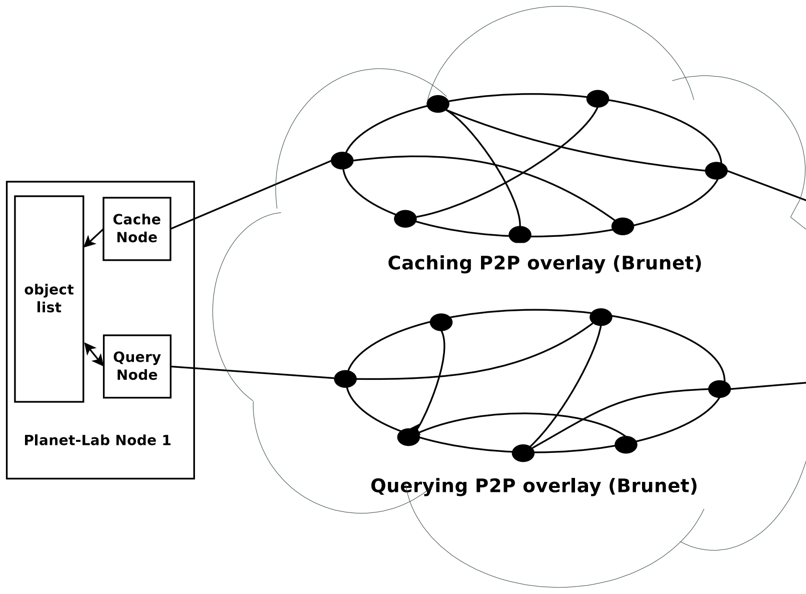


Fig. 23. P2P (Brunet) network overview

B. Brunet-Network Topology

The Brunet P2P library provides mechanisms for building and maintaining structured P2P networks of overlay nodes. The structure of Brunet is the same as Symphony's structure, in which each node maintains connections with its nearest-neighbor nodes in P2P address space. These connections are called `structured near connections`. In addition to the structured near connections, each node also connects to k distant nodes. These nodes are called `structured shortcut connections`, and nodes at distance d with proportional to $\frac{1}{d}$ are selected as shortcut connections. The shortcut connections contribute to the decreased access time for bounded broadcasting. With the help of shortcuts, the hop distance in a given range of bounded broadcasting is limited to $O(\frac{1}{k}\log^2 n)$, given a network of n nodes [27]. Typically $k \approx \log n$, so time is $O(\log n)$.

The Deetoo service creates two virtual nodes on one physical overlay node. Each virtual node joins either a caching network or a querying network, and they share a list of cached objects. By receiving a caching message, a caching node writes an object's information in the cache list. Caching nodes also conduct stabilization as connections change and as network size changes. The variation of network size estimation results in the change of the broadcasting range of cached objects. Thus, stabilization is required to maintain the correct number of replicas of each object. The querying node is responsible for responding to query messages. Upon receiving a query message, the node first reads the query type and then executes a proper query resolving action. Figure 23 shows how two virtual networks are built on top of Brunet.

C. Network Size Estimation

The accuracy of network size estimation is important in distributed systems such as P2P networks. In Deetoo's bounded broadcasting, the performance of Deetoo's search algorithm depends largely on the accuracy of estimation because the range of bounded-broadcasting is determined by the estimated network size. To address the issue of network size estimation, a sequential update is introduced. First, a node calculates node density using only direct neighbors (one left and one right connection) after it joins a network. Based on the node density, the node gets the first estimation, which is N_0. In the next estimation step, the node sends a simple ping message to a $\log N_0$-hop away node and retrieves the remote node's address information; this gives N_1. Finally, it requests the estimation to its shortcut neighbors then takes a median. For both the caching and querying purposes, the ranges for bounded-broadcasting are determined by the final median estimation. The accuracy of the sequential estimation is compared with the actual number of nodes in the network in Section VII-A.

D. MapReduce-Based Cache/Query System

Map and reduce functions are popular paradigms in functional programming languages (e.g., Ruby, Python, and LISP). A map function transforms each element (x_i) in a collection ($x_1, x_2 ... x_n$) into a different key element (y_i), thus creating a collection of intermediate elements ($y_1, y_2 ... y_n$). A reduce function computes an aggregation (z) over this collection. Based on the concept of the map and reduce functional programming model, software frameworks (e.g., Hadoop [34] and Google [35]) have been developed to efficiently parallelize computations on large datasets. The map function usually works on (key/value) pairs to create intermediate (key/value) results. Reduce functions work on intermediate (key/value) results while aggregating intermediate values associated with the same intermediate keys. The framework distributes map and reduce tasks among nodes in a cluster to enable parallel processing, while users only have to specify the appropriate map and reduce functions associated with their computation without having to worry about details of distributed parallel job execution.

The implementation of Deetoo on top of Brunet consists of five functional modules. They include a P2P network module, a MapReduce core, a tree generation module, a map module and a reduce module. Figure 24 shows the architecture of MapReduce function modules. When a task (caching/querying) is assigned through the underlying P2P network's Remote Procedure Call (RPC), the MapReduce core module first generates a tree using bounded broadcasting. After a tree is successfully generated, map functions are executed in parallel at all the nodes in the tree. The MapReduce core in an initiating node starts the map-reduce process. While Map is processing its own function at a node, the node waits for the response from its child nodes. Upon receiving results from child nodes, the node reduces the job, then returns the reduced result to its parent node. When reduce is completed, the whole result is passed to the root node. Note that the root node in a tree is the initiating node for cache or query. Figure 25 illustrates a simple example of a MapReduce tree. x_i is an argument for a map function. z_j and the result of map are arguments for a reduce function. Let a map function be $f(x_i)$, a result of reduce be z_i, and a reduce function

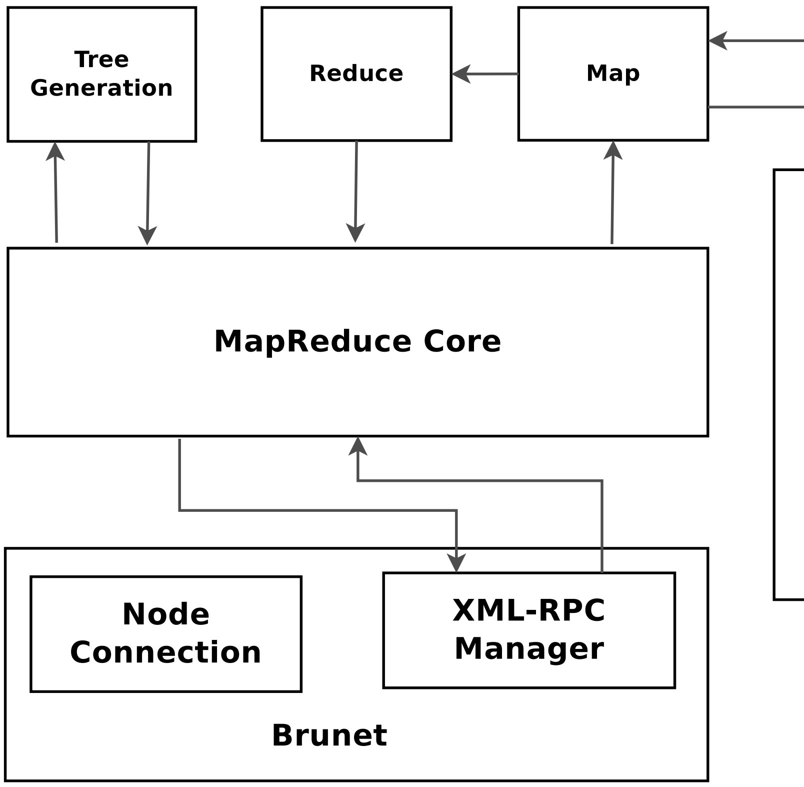


Fig. 24. MapReduce function modules for query system

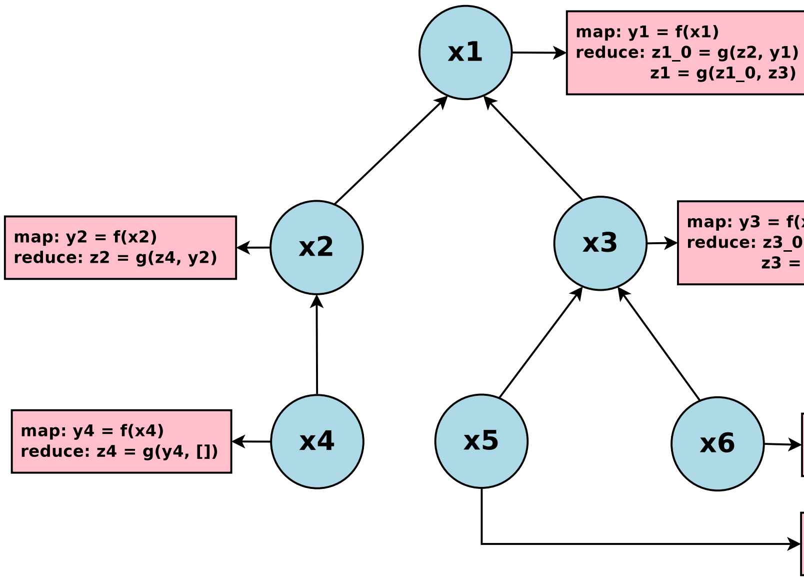


Fig. 25. Simple example of map-reduce tree

be $g(z_j, .., z_k, x_i)$. The root node, x_1, executes its own map function and gets a result, $f(x_1)$. The child nodes x_2 and x_3 pass their reduce results. The map and reduce functions are executed simultaneously in the nodes at the same level. Finally, x_1 receives reduce results from x_2 and x_3 and it reduces with them and its own map result. The reduce results in leaf nodes (x_4, x_5, x_6), which are the same as their map results in this example because they do not have any child nodes that pass any results.

The details of each module's functions is described in the following sections.

1) P2P network module: The P2P network module is responsible for handling new node joins and departures, connection management with neighboring nodes, and message routing. The current implementation of Deetoo is developed on top of Brunet. While the underlying network module for the Deetoo implementation is Brunet, Deetoo can be built on many other P2P platforms, such as Chord [4], CAN [5], and Pastry [36]. This can be done by building a 2-dimensional space, then cutting the space to intersect at one point for cache and query.

2) MapReduce core module: The MapReduce core module is responsible for distributing map and reduce functions using bounded broadcast. When a user initiates a MapReduce task, the request is conveyed to the MapReduce core module through the underlying P2P network's Remote Procedure Call (RPC) module. The MapReduce core module checks the map arguments, the reduce arguments, and the broadcast region arguments. The map and reduce arguments will be passed to the map and reduce functions, respectively. Map arguments include an object to be inserted into a network in case of caching and an object and a query type to be passed as map arguments for querying. The results from child nodes and the result of map are passed as reduce arguments. The broadcast argument describes a bounded-broadcast region that the node is responsible for. The MapReduce core module disseminates the task to nodes which reside under its responsible region after manipulating the broadcast region argument appropriately.

3) Generating a tree module: The generating tree functional module makes use of bounded broadcast to replicate objects as well as to propagate query messages. With the existence of shortcut connections in a Brunet P2P network, the bounded broadcast builds a tree whose depth is short ($\approx \log N$) compared to a naive ring-based P2P overlay. The detailed operation of bounded broadcast that builds a multicast tree is described in [37]. The range of bounded broadcasting is selected at random, while the size of the range depends on the estimated network size and the replication factor, α. The replication factor is a user-defined parameter to the generating tree functional module. By adjusting the replication factor, users are able to control the number of replicas in a caching network or manage query success probability in a querying network.

4) Mapping module: There are two separate map functions for caching and querying for the test. For caching, the map function inserts a data object into the node's data list and returns true if data insertion was successful. In the querying purpose, map function searches a specified string from the data list. The map function for the query also supports regular expression search using the existing .Net library. The result of the map function is either success or failure if it is called for the caching purpose, while the map function for the querying returns the matching object. The map result is transferred to the node's reduce function module as an argument. For the caching and the querying purposes, map functions are described as follows:

Cache Map Function: The pseudo-code for a cache map function is shown in Algorithm 1. The function takes an object to be inserted as argument. The node which runs the map function accesses its list of objects and checkes if the inserting object already exists in the list or not. The result of cache map function returns `true` or `false` depending on success or failure of insertion.

Algorithm 1 CacheMap

```
Require: $object\ o$ {an object to be inserted}
Require: $Node\ n$
1: $object\_list \leftarrow n.getObjects()$
2: if $o$ is not in $object\_list$ then
3:     $object\_list.put(o)$
4:     return true
5: else
6:     return false
7: end if
```

Query Map Function: Algorithm 2 describes the query map function. The function requires an object to be searched as an argument and it returns a list of matching objects. If the function cannot find any match, the query map function returns an empty list.

Algorithm 2 QueryMap

```
Require: $object\ o$ {an object to be searched}
Require: $Node\ n$
1: $object\_list \leftarrow n.getObjects()$
2: $result = [\ ]$
3: for all $v \in object\_list$ do
4:     if $o$ matches $v$ then
5:         $result.Add(v)$
6:     end if
7: end for
8: return $result$
```

5) Reducing module: In the reduce functional module, the map results from the working node and the reduce results from the child nodes are passed to the reduce module as inputs. Assume that a node has n child nodes in a tree. The working node sends a message to its child nodes in the tree that is built using bounded broadcast. Thus, the node expects n results from its child nodes and one result from its map module. By reducing n reduce results and one map result, the node transfers $n+1$ reduce results to its parent node in the tree. In the caching process, the boolean results which indicate success or failure of caching, the number of nodes and the time spent in terms of tree depth are passed as arguments to the reduce function. In the querying process, the boolean results are replaced by the matching results. The reduce function for regular expression match concatenates all the reduce results and passes the results up the tree. By the end of each process over the network, a user collects information about the total number of nodes visited and tree depth in addition to caching or query result. As reduce is processed at each node in a tree, the results are propagated back through the tree. For the caching and the querying purposes, reduce functions are described as follows:

Cache Reduce Function: Algorithm 3 explains how a cache reduce works. When a cache reduce is called by a node, it collects all the results from the node's children in a tree and they are transferred to the cache reduce function as arguments. In the test setting, the child results are a list of boolean values that indicate the success or failure. In addition, the result of map from the node is passed as a reduce argument. If one of the arguments is $NULL$, the reduce function returns only a valid result from either map or child result. The function combines the map result with the results from child nodes, then return the combined result.

Algorithm 3 CacheReduce

```
Require: Map Result $m$, {[true, false,..,false]}
Require: Child Result $c$ {[false,true,..,true]}
1: $result\_list \leftarrow [\ ]$
2: if $m$ is NULL then
3:         $result \leftarrow c$
4: else if $c$ is NULL then
5:         $result \leftarrow m$
6: else
7:         $result \leftarrow m.combine(c)$
8: end if
9: return $result\_list$
```

Query Reduce function: The pseudo-code for the query reduce function is shown in Algorithm 4. This is almost the same as the cache reduce function. The difference is that the result from both child node and map is a list matching object.

Algorithm 4 QueryReduce

```
Require: Map Result $m$, {a list of matching objects}
Require: Child Result $c$ {a list of matching objects}
1: $result\_list \leftarrow [\ ]$
2: if $m$ is NULL then
3:         $result \leftarrow c$
4: else if $c$ is NULL then
5:         $result \leftarrow m$
6: else
7:         $result \leftarrow m.combine(c)$
8: end if
9: return $result\_list$
```

VII. Experimental Environment

In this section, the performance of Deetoo is confirmed via experiments in Planet-Lab. Deetoo nodes, each composed of two virtual Brunet nodes, were deployed. The virtual nodes form two different virtual overlay networks, one for caching, the other for querying. Brunet framework manages connections among nodes and provides the mechanism for UDP packet transport. Also, it supports NAT and firewall traversals. The Deetoo search system was implemented on more than 400 Planet-Lab nodes. For the experiment, 100 unique string objects were cached over the network. Then, the querying process was repeated 100 times for each unique object.

Test was performed with two different values of replication factor (α) to observe the effect of increasing or decreasing the caching or querying range size. The replication factor in the experiments was set to 1.0 and 3.0. Note that the replication factor is the only parameter to affect a hit rate, which is $e^{-\alpha}$. $< \alpha >$ nodes get both cache and query.

The matching function used regular expression match for the entire test. All caching and querying processes were initiated on randomly selected nodes and messages were broadcasted within randomly chosen ranges.

A. Evaluation and Results

The evaluation is focused on (a) accuracy of network size estimation, (b) query success probability, (c) latency in terms of the number of the depth in a tree generated by bounded broadcast, (d) communication cost. For each metric, moving averages in the time span of 1000 trials are calculated.

1) Network Size Estimation: In the test, the moving average of sample size 1000 is used for evaluation. In other words, past 1000 results are averaged at the time indicated. The moving average is good for showing the variation of small subset of results given a very large number of samples. Figure 26 shows how accurate the network size estimator performed. The results of network size estimation was slightly larger than the actual network size. However, the deviation was very small. The difference between measured network size in each node and the actual network size is resulted from the non-uniform distribution of nodes‘ addresses.

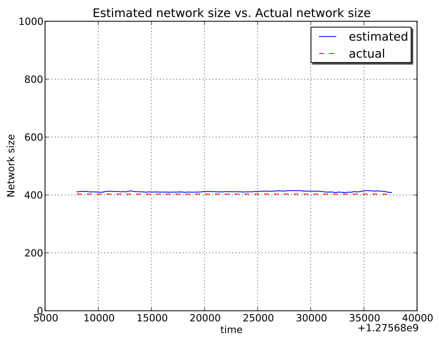


Fig. 26. Estimated Network Size

2) Query Success Probability: The query success probability was measured. The first figure in Figure 27 shows the query success probability as a function of measured time with 1.0 of replication factor and the other figure in Figure 27 shows the probability when a replication factor is set to 3.0. The variation is observed in each experiment. The query success probability is only depends on the replication factor and the network size. Considering a replication factor is deterministic as a user can specify it according to a user‘s desired performance, the network size is the most and the only influential factor for the query success probability. As a nature of distributed network with no global information, Each result was compared to the theoretical result that is approximately $1 - \exp^{-\alpha}$.

As seen in the figure, the query success probability is almost constant regardless of the network size assuming that the estimation is correct. The results were comparable to the theoretical results. The constant success probability is desired since Deetoo can perform the search with preferred success probability by adjusting broadcast range with different replication factor. As expected, the larger the replication factor, the higher the query success probability.

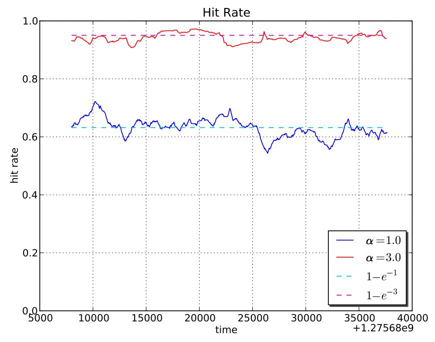


Fig. 27. Query Success Probability

3) Latency: The latency in the test is measured by counting tree depth. In the calculation of latency, the number of out-of-range links before finding a node in the range by greedy routing is added to the tree depth. We assumed that per-hop latency between any two nodes is all the same. According to [37], the latency of bounded braodcast in a given network size, n, is bounded by $O(\frac{1}{k}\log^2 n)$, where k is the number of shortcut links. In Brunet, each node maintains approximately $\log n$ shortcut connections. Thus, the tree depth is $O(\log^n)$. The Figure 28 compares theoretical results with measured latency when α is 1.0 and 3.0, respectively. The experimental results shows shorter latency because the number of shortcut connection is more in Brunet than that of calculation. Considering that MapReduce is executed in parallel in a tree, a shorter latency produces a more desirable responsiveness. The actual latency(L) for the querying is measured. The latency between a pair of nodes is not measured but assuming that the per-hop latency (L_{hop}) between two nodes is constant, per-hop latency can be calculated from the total latency and the depth of tree(l_D).

$$L_{hop} = \frac{L}{l_D}$$

For $\alpha = 1.0$, the latency will be increased to $9.42(sec)$ (per-hop latency $\times$ communication cost) if a query message is delivered sequentially instead of using bounded broadcast. Sequential message relay takes 4 times more time than bounded broadcast.

4) Communication Cost: Communication cost is evaluated as the number of nodes reached by a caching or a querying message. In the experiment, only 5% of nodes needed to be accessed by searching process to achieve over 60% of query success probability and less than 9% nodes were visited to satisfy over 90% of query success probability. Since the size of UDP packet for a string object caching is 384 $byte$

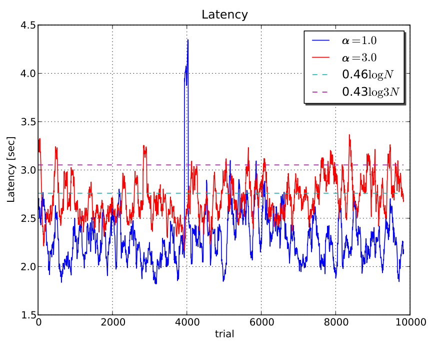


Fig. 28. Measured Latency

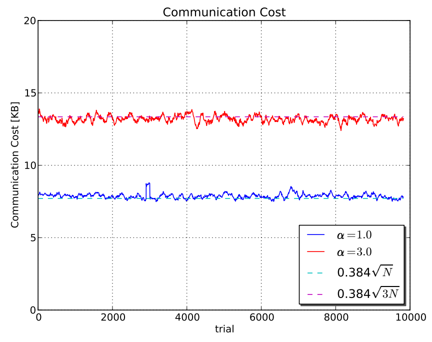


Fig. 29. Total bytes transferred for each query

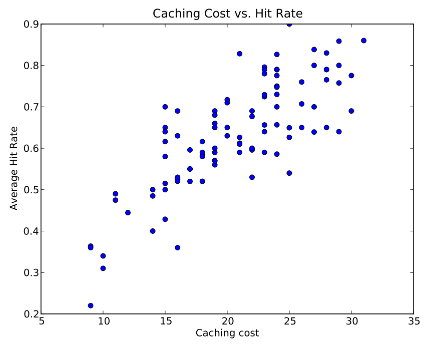
and the average number of messages for $\alpha = 1.0$ is 20.54, approximately 8 KB of total data is transferred for a single cache. Figure 29 illustrates the total bytes transferred in the network for each query.

5) Replication vs. Success Probability: There exists a trade-off between the number of replicas per object and query success probability. The test counted the number of replicas per each unique objects after stabilization. For the query success probability, we counted the number of successfully resolved query per each object and averaged. The Figure 30 shows the trade-off between replication and query success probability with 1.0 of α and 3.0, respectively. We observed that increment of query success probability as the number of replica increased until query success probability reached 100/

6) Load Balancing: Let m be the number of objects cached, N be the number of nodes, and C be the communication cost. It is known that $C = \sqrt{\alpha N}$ from the equations (3) and (4) in Section I. Thus, the number of objects per node, m_{ave}, is:

$$m_{ave} = \frac{m \times C}{N} \quad (25)$$
$$= m\sqrt{\frac{\alpha}{N}} \quad (26)$$

I measured the number of objects per node from 368 PlanetLab nodes (N). 500 objects (m) were cached over the network with a replication factor 1. As seen in Figures 31 and 32, the load is evenly distributed. The average number of objects in a



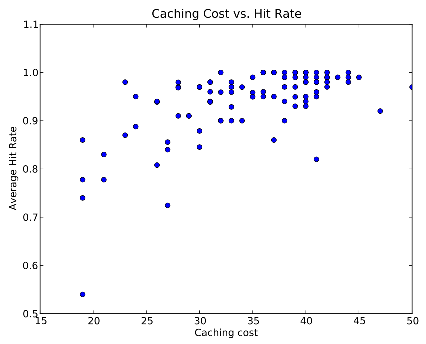


Fig. 30. Communication Cost for Caching vs. Query Success Probability

node is 24.5 and the standard deviation is 5.46. The theoretical expectation for the number of replicated objects per node is 26.06 if the replication factor is set to 1. The reason for the mismatch between the measured result and the theoretical result came from the over estimation of the network size. The average of estimated size was 395.6, but the actual size was 384. The over-estimation resulted in the smaller number of replicated objects.

VIII. Conclusion and Future Work

In this paper, we introduced Deetoo, a scalable routing peer-to-peer protocol for unstructured networks which provides efficient caching and lookup functionality with $O(1)$ query hit-rate, $O(\sqrt{N})$ replication, $O(\sqrt{N})$ query cost, and $O(\log^2 N)$ search time. Deetoo, allows objects to be updated and deleted, and can be used to handle general queries. Specifically, any problem that can be mapped onto selecting (with high probability) objects which match some query can be run over Deetoo. The performance of Deetoo is verified by theoretical analysis as well as simulation. We confirmed that the simulation results matches theoretical results.

In addition, Deetoo search technique is evaluated and tested on Planet-Lab to verify performance Deetoo in real P2P network environment. Deetoo search module is developed on the existing P2P overlay. The test of The regular expression search confirmed Dootoo‘s general type of query resolving ability. The test results show that Deetoo's caching and querying performance is very close to both theoretical results and simulation results. The performance of Deetoo on real network shows that Deetoo is an attractive solution for unstructured search on top of structured P2P networks. The developed Deetoo search component on top of P2P substrate which is Brunet in this work is reusable to other existing P2P overlay networks.

References

[1] M. K. Kiran Nagaraja, Sami Rollins, "From the editors: Peer-to-peer community–looking beyond the legacy of napster and gnutella," *IEEE Distributed Systems Online*, vol. 7, no. 3, 2006.

[2] J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective," in *Proc. of the 32nd ACM Symp. on Theory of Computing*, 2000.

[3] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proc. of the 16-th Int'l Conf. on Supercomputing (ICS-02)*. New York: ACM Press, 2002, pp. 84–95.

[4] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. on Networking.*, vol. 11, no. 1, pp. 17–32, 2003.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. of ACM SIGCOMM 2001*, 2001. [Online]. Available: citeseer.ist.psu.edu/ratnasamy01scalable.html

[6] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A global-scale overlay for rapid service deployment," *IEEE J. on Selected Areas in Comm.*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[7] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed hashing in a small world," *Proc. 4-th USENIX Symp. on Internet Technologies and Systems*, pp. 127–140, 2003. [Online]. Available: http://www-db.stanford.edu/ manku/papers/03usits-symphony.pdf

[8] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays," in *Symp. on Networked Systems Design and Implementation*. USENIX, 2004, pp. 99–112.

[9] C. Tang, Z. Xu, and S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in *Proc. of the ACM SIGCOMM 2003 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Comm.*, A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, Eds. Karlsruhe, Germany: ACM, 2003, pp. 175–186.

[10] E. G. S. Bernard Wong, Aleksandrs Silvkins, "Approximate matching for peer-to-peer overlays with cubit," Computing and Information Science, Cornell University, Tech. Rep., 2008.

[11] J. Albrecht, D. Patterson, and A. Vahdat, "Distributed resource discovery on planetlab with sword," in *WORLDS*, 2004.

[12] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," in *SIGCOMM*, 2004, pp. 353–366.

[13] M. Cai, M. Frank, J. Chen, and P. Szekely, "Maan: A multi-attribute addressable network for grid information services," in *J. of Grid Computing*. IEEE Computer Society, 2003, p. 184.

[14] "Kazaa," http://www.kazaa.com/, 2010.

[15] S. Johnstone, P. Sage, and P. Milligan, "iXChange - A self-organising super peer network model," in *IEEE Symp. on Computers and Comm.*, vol. 00. Washington, DC, USA: IEEE Computer Society, 2005, pp. 164–169.

[16] L. Guo, S. Jiang, L. Xiao, and X. Zhang, "Exploiting content localities for efficient search in P2P systems," in *Int'l Symp. on Distributed Computing*. Springer, 2004, pp. 349–364.

[17] ——, "Fast and low-cost search schemes by exploiting localities in P2P networks," *J. of Parallel and Distributed Computing*, vol. 65, no. 6, pp. 729–742, 2005.

[18] K. Sripanidkulchai, B. M. Maggs, and H. Zhang, "Efficient content location using interest-based locality in peer-to-peer systems," in *INFOCOM*. IEEE, 2003.

[19] S. Jiang, L. Guo, X. Zhang, and H. Wang, "LightFlood: Minimizing redundant messages and maximizing scope of peer-to-peer search," *IEEE*

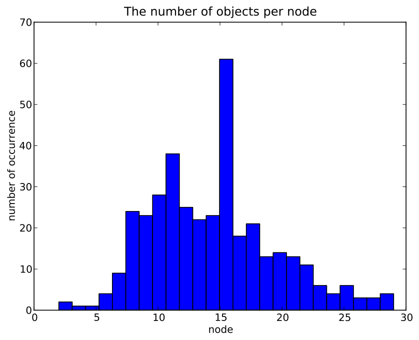


Fig. 31. Load distribution: originally 220 objects were cached over the entire network.

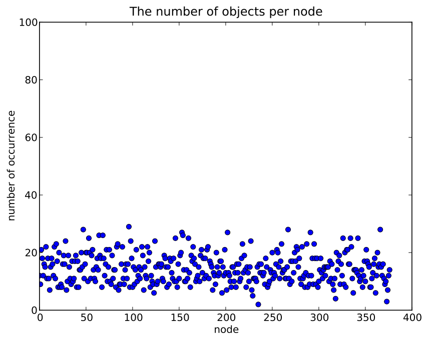


Fig. 32. The number of Cached Objects per node: 220 objects are cached over the entire network.

Fig. 33. Load Balancing

Trans. on Parallel and Distributed Systems, vol. 19, no. 5, pp. 601–614, 2008.

[20] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, "Search in power-law networks," *Physical Rev. E*, vol. 64, p. 46135, 2001.

[21] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," *SIGCOMM Computer Comm. Rev*, vol. 32, no. 4, pp. 177–190, 2002.

[22] B. F. Cooper, "Quickly routing searches without having to move content," in *4-th Int'l Workshop on Peer-to-Peer Systems*, ser. Lecture Notes in Computer Science, vol. 3640. Springer, 2005, pp. 163–172.

[23] K. S. Ming Zhong, "Popularity-biased random walks for peer-to-peer search under the square-root principle," in *Int'l Workshop on Peer-to-Peer Systems*, 2006.

[24] N. Sarshar, P. Boykin, and V. Roychowdhury, "Percolation search in power law networks: Making unstructured peer-to-peer networks scalable," in *Proc. of the 4-th IEEE int'l conf. on peer-to-peer computing*, 2004.

[25] X. Liu, Q. Huang, and Y. Zhang, "Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks," in *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems (SenSys '04)*. Baltimore, MD, USA: ACM, Nov. 2004, pp. 122–133.

[26] M. Naor and U. Wieder, "Scalable and dynamic quorum systems," *Distributed Computing*, vol. 17, no. 4, pp. 311–322, May 2005.

[27] J. Kleinberg, "Small-world phenomena and the dynamics of information," in *Advances in Neural Information Processing systems(NIPS)*. cambridge, MA: MIT Press, 2001, no. 14.

[28] "Brunet," http://boykin.acis.ufl.edu/wiki/index.php/Brunet, P. Oscar Boykin, 2010.

[29] "Planet-lab," http://www.planet-lab.org, Princeton University, 2007.

[30] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo, "Ip over p2p: Enabling self-configuring virtual ip networks for grid computing," in *In Proc. of 20-th Int'l Parallel and Distributed Processing Symp. (IPDPS-2006)*, 2006, pp. 1–10.

[31] A. Ganguly, A. Agrawal, P. O. Boykin, and R. J. Figueiredo, "Wow: Self-organizing wide area overlay networks of virtual workstations," *J. of Grid Computing*, vol. 5, no. 2, pp. 151–172, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10723-007-9076-6

[32] D. Wolinsky and R. Figueiredo, "Simplifying resource sharing in voluntary grid computing with the grid appliance," in *IEEE Int'l Symp. on Parallel and Distributed Processing, (IPDPS '08)*, 2008, pp. 1–8.

[33] R. J. Figueiredo, P. O. Boykin, P. S. Juste, and D. Wolinsky, "Integrating overlay and social networks for seamless p2p networking," in *Proc. of the 2008 IEEE 17-th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 93–98.

[34] http://hadoop.apache.org/, The Apache Software Foundation, 2007.

[35] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[36] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001, pp. 329–350.

[37] T. W. Choi and P. O. Boykin, "Deetoo: Scalable unstructured search built on a structured overlay." in *7-th Int'l Workshop on Hot Topics in Peer-to-Peer Systems*, 2010.