YAPPERS: A Peer-to-Peer Lookup Service over Arbitrary Topology

Prasanna Ganesan Qixiang Sun Hector Garcia-Molina
Computer Science Department
Stanford University, Stanford, CA 94305, USA
{prasannag, qsun, hector}@cs.stanford.edu

***Abstract*— Existing peer-to-peer search networks generally fall into two categories: Gnutella-style systems that use arbitrary topology and rely on controlled flooding for search, and systems that explicitly build an underlying topology to efficiently support a distributed hash table (DHT). In this paper, we propose a hybrid scheme for building a peer-to-peer lookup service over arbitrary network topology. Specifically, for each node in the search network, we build a small DHT consisting of nearby nodes and then provide an intelligent search mechanism that can traverse all the small DHTs. Our hybrid approach can reduce the nodes contacted for a lookup by an order of magnitude compared to Gnutella, allows rapid searching of nearby nodes through quick fan-out, does not reorganize the underlying overlay, and isolates the effect of topology changes to small areas for better scalability and stability.**

Methods Keywords: System Design, Peer-to-Peer

I. Introduction and Related Work

We study the problem of building a *peer-to-peer lookup service* on top of an arbitrary dynamic peer-to-peer (P2P) overlay network over which we have no control. A lookup service is one that maintains a dynamic set of key-value associations (usually multiple values for a single key), and permits queries that request values associated with a key. A query that requests all the values associated with a key is a *total lookup*. Similarly, a query that requests some values for a key is a *partial lookup*.

Traditionally, there have been two flavors of P2P lookup services. The first kind consists of systems like Gnutella[1] and FastTrack[2] that do not organize the content in the network. If a peer node has some $\langle key, value\rangle$ pairs it wants to "insert", it simply stores them itself. Consequently, answering a total lookup (i.e., getting all the values of a key) requires flooding the entire network to search every node. In practice, queries are treated as partial lookups that don't need retrieval of all the available answers. Since there are usually a large number of answers available for each query, the search radius can be limited and the query reaches only a portion of the entire network.

These Gnutella-style networks are deployed in real life because of their simplicity and the lack of complex state information at each peer. When a node goes down, the system loses the $\langle key, value\rangle$ associations that this node "inserted" and nothing else, which is perfectly acceptable. Hence, frequent network topology changes have very little impact on performance. Moreover, these systems can operate on any type of P2P overlay network, thus allowing others to construct an "optimal" overlay. For example, Raghavan suggested a low diameter overlay with high connectivity in [3]; Cohen suggested constructing overlays where nodes with similar content are neighbors in [4]. When there are significant bandwidth differences between clients, Morpheus [5] (a client using FastTrack) changes the topology so that high bandwidth super-peers are at the core of the overlay and low bandwidth clients are on the edge of the overlay.

The flooding-based search algorithm in Gnutella-style networks can also be replaced by more efficient alternatives. Among the suggested algorithms are the iterative deepening technique to slowly increase the flooding radius [6], using random walks instead of radial flooding for search [7], and aggregating and distributing local search indices to guide search direction [8].

In contrast to Gnutella-style networks, the second kind of systems such as CAN [9], Chord [10], Pastry [11] and Tapestry [12] build a distributed hash table (DHT) on top of the overlay to provide efficient querying. In DHTs, keys are hashed into a keyspace, and each peer is assigned a small segment of this space. Therefore, a lookup request for a key simply entails finding the peer responsible for the key's hash value. Notice that, by the construction of the DHT, there is no distinction in performance between partial and total lookups since there is one node that has all the values for a key. So if there are many clients interested in doing partial lookups on a particular key, then DHT-based systems cannot take advantage of the partial lookup and might have a hot-spot problem where the one node responsible for the key is overloaded. Hence, DHTs are more suited to searching for rare items (a key with very few values) than popular items (a key with huge number of values).

In building DHTs, these systems have to impose strict constraints on the overlay topology to guarantee efficiency in their search protocol. For instance, CAN imposes a d-dimensional torus structure while Chord creates a ring with long-distance hop pointers. Since key-value pairs are not stored locally at each node, special care is needed to preserve data when nodes leave the system. Moreover, when nodes join and leave frequently, it can affect state information at a significant number of nodes to maintain the strict topology.

Given the advantages and disadvantages of the Gnutella-style and DHT-based systems, our objective is to design a hybrid system, YAPPERS (Yet Another Peer-to-PEeR System), that operates on top of an arbitrary overlay network, just as

Gnutella does, while providing DHT-like search efficiency. There are many applications where such operation on an arbitrary overlay network is either critical or can come in handy. For example, one might wish to utilize an overlay that maps closely to the underlying wired or wireless network layer in order to maximize messaging efficiency. Another interesting application, especially with the increasing focus on wireless roaming internet access, is the use of P2P for providing local directory information. In this case, the system needs to support queries requiring answers that are *near* the source of the query, with the notion of nearness being dictated by the location of content on the overlay network itself. Applications on social networks, where a node can interact only with its *friends*, again require the ability to work with the given overlay and do not afford the lookup service the luxury of building its own overlay network. We examine these and other applications in greater detail in [13].

Our four main design goals in designing our lookup service are:

1) impose no constraints on topology.
2) optimize for partial lookups where there are many values for a key and clients are satisfied by a small subset of the values.
3) contact only nodes that can contribute to the search result rather than flooding blindly.
4) minimize the effect of topology changes so that the maintenance overhead is independent of the overlay size.

In the remainder of the paper, we first state our assumptions and give an overview of YAPPERS in Sections II and III. We then present the details of our algorithm that meets the above design goals in Section IV. In particular, we prove the correctness of our algorithm and discuss how to handle dynamic node arrivals and departures. We also evaluate the performance of YAPPERS on a Gnutella-network snapshot and synthetic regular graphs in Section VI.

II. Problem Definition and Assumptions

We assume the following model for each of the peer nodes:

- When a node is created, there is a third-party network layer that determines a set of live nodes as its new neighbors in the overlay.
- Each node "owns" a (possibly empty) set of $\langle key, value \rangle$ pairs. When the node joins the network, it registers these pairs with the lookup service. The node may choose to register additional pairs with the service, or delete some registered pairs at any point in time.
- A node may initiate either partial-lookup or total-lookup queries at any point in time.
- When a node leaves the system, it may or may not leave gracefully.
- When a node leaves the system, the $\langle key, value \rangle$ pairs that it "owns" do not need to be preserved in the system.
- A node may establish connections to other nodes directly, even if they are not neighbors in the overlay network.

The final assumption is not strictly necessary but helps in improving the efficiency of the system. With these assumptions, let S be the set of $\langle key, value \rangle$ pairs registered with the lookup service. We define:

- $TotalLookup(N, k)$ as the set of values returned by the service for a total-lookup query on key k originating at node N.
- $PartialLookup(N, k, n)$ as the set of n values returned by the service for a partial-lookup query on key k originating at node N. If the total number of values for k is less than n, the partial lookup is defined to be equivalent to the total lookup.
- $Values(k) = \{v | \langle k, v \rangle \in S\}$

Then the lookup service is correct if and only if:

1) $TotalLookup(N, k) = Values(k)$ for all nodes N and keys k.
2) $PartialLookup(N, k, n) \subseteq Values(k)$ and $|PartialLookup(N, k, n)| \geq \min(n, |Values(k)|)$ for all nodes N, keys k and integers n.

When nodes join or leave, we do allow temporary inconsistency in lookup results while their presence in the network are being updated.

III. Overview of YAPPERS

Intuitively, YAPPERS works as follows: The keyspace of all the keys that need to be stored is partitioned into a small number of buckets. For ease of exposition, consider an example where the keyspace is divided into two buckets. Let us say that keys are either white or gray. Every node in the network is also assigned a color, white or gray, based on some criterion. For example, we could hash the IP address of a node to determine whether it should be white or gray. Consider the white node A in Figure 1. If it wants to register a value for a white key $\langle k_w, v_1 \rangle$, this pair can be stored at A itself, since A is also white. If on the other hand, node A wants to register a gray key and its value $\langle k_g, v_2 \rangle$, then A looks for a neighboring gray node. In this case, its neighbor B is a gray node. So, A stores this pair at node B.

Now, consider what happens at query time. A query for a gray key needs to be forwarded only to gray nodes in the network, and a query for a white key only to the white nodes. In order to exploit the partitioning of the data, we need some way of confining queries on gray (resp., white) keys to the gray (resp., white) nodes in the network. If the query originates at a white node, we forward the query to a gray neighbor of the node. Notice that some white nodes in the network might not have any gray neighbors at all. In Figure 1, if A's neighbor B had also turned out to be white, A wouldn't have a gray neighbor. In the next section, we explain how to avoid this problem by expanding neighborhoods and assigning multiple colors to nodes. Once the query makes its way into one gray node, we forward the query to all the gray nodes that the current node knows about. To guarantee that gray queries end up being forwarded to all the gray nodes, we require that each node knows all nodes within 3 hops of it, and forwards a gray query to all the gray nodes it knows about in this 3-hop radius. An example of a search for a gray key is shown in Figure 1. The search starts at the white node A, which forwards the query to one of its gray neighbors, B. Node B then forwards

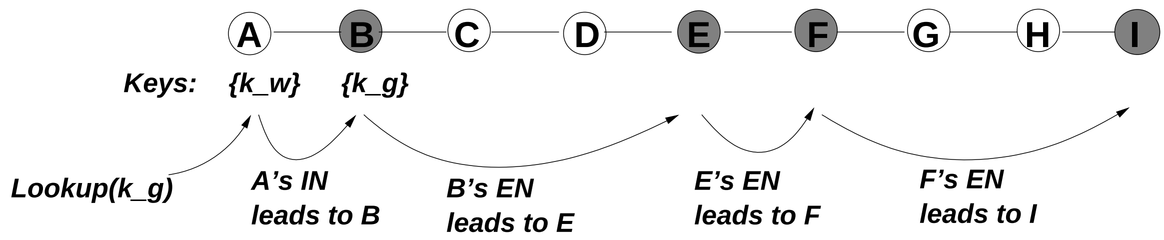

Fig. 1. Using both the immediate (denoted by IN) and the extended (denoted by EN) neighborhood information, a lookup for key k_g at node A can reach all nodes that stores key k_g.

it to all the gray nodes that it knows within 3 hops, and the process continues. Our protocols ensure that this forwarding terminates only after all the gray nodes have been reached.

More generally, if we permit a node to store a $\langle key, value\rangle$ pair at an appropriately-colored node within h hops of it (instead of within 1 hop of it), it is sufficient for a node to know all its neighbors within $(2h + 1)$ hops in order to guarantee that we can exhaustively hop through all the gray nodes without touching any white node. This statement holds when we have nodes and keys of any number of colors, not just for the simple case of 2 colors. We call the nodes within h hops the *immediate neighborhood* of a node, and the nodes within $2h + 1$ hops the *extended neighborhood* of the node. We discuss the concept of the immediate neighborhood in Section IV-A and the extended neighborhood in Section IV-B. We then proceed to define the search protocol based on these concepts and prove that the protocol is correct. Section IV-C discusses node arrivals and departures.

IV. Basic Algorithm

YAPPERS divides a large overlay network into many small and overlapping neighborhoods (the immediate neighborhoods). The data within each neighborhood is partitioned among the neighbors like a distributed hash table. When a lookup occurs and the neighborhood cannot satisfy the request, YAPPERS intelligently forwards the request to nearby neighborhoods, or the entire network if necessary. These forwardings require each node to know a larger set of nodes (the extended neighborhood) that covers its neighbors' neighbors.

A. The Immediate Neighborhood

The immediate neighborhood of a node A, denoted by $IN(A)$, is the set of nodes where A may store its $\langle key, value\rangle$ pairs. In managing the immediate neighborhood $IN(A)$, we need to address two questions:

1) Given a node A, which nodes should be included in $IN(A)$?
2) Given $IN(A)$, how do we partition the key space into multiple colors and assign each color to nodes in $IN(A)$?

Moreover, the solution must strive to maintain the following two useful characteristics:

- *Consistency*: If a node X is in two different neighborhoods $IN(A)$ and $IN(B)$, both A and B assign the same color to node X.
- *Stability*: If a node X is in $IN(A)$, then X is assigned the same color regardless how $IN(A)$ changes dynamically when nodes enter or leave the system.

Both characteristics are desirable because they improve the overall system efficiency. Consistency avoids costly synchronizations among nearby nodes to determine which nodes have which colors. Stability reduces data relocation when the underlying overlay network changes.

With these desired characteristics in mind, YAPPERS addresses the first question by defining the immediate neighborhood of node A, $IN(A)$, in the overlay network G as

- $IN(A) = \{v \mid \delta_G(v, A) \leq h\}$ where δ_G returns the minimum distance between two nodes in G. In other words, $IN(A)$ contains all nodes within h hops of A in the overlay network, including node A itself.

Specifically, our YAPPERS implementation uses $h = 2$. We chose a small immediate neighborhood in order to provide long-term stability to the system. If the immediate neighborhood is large, then frequent node arrivals and departures within the network will incur large overheads in maintaining an accurate view of the immediate neighborhood and reduce the efficiency of searches when the view is incorrect. To make this observation more concrete, consider the Chord network. In essence, Chord has a single immediate neighborhood that contains every node in the network. However, lookup efficiency in Chord will degrade when nodes enter and leave the network frequently because the system takes a long time to reach a stable configuration where all the finger pointers, used by the lookup process, are accurate.

Addressing the second question, YAPPERS partitions the key space among the nodes in $IN(A)$ using the hash values of the node IP addresses. Formally,

- a node X with IP address IP_X is assigned key k if $HASH(k) \equiv (HASH(IP_X) \bmod b)$ where b is the number of distinct hash buckets we use.

In other words, the keyspace is divided into b hash buckets, or b different colors $C_0, C_1, \ldots, C_{b-1}$. We say that a key k is of color C (or hashes to color C) if the hash function assigns k to bucket C. If a node IP address hashes to bucket C, we say the node is of color C. Note that by using nodes' IP addresses, the partitioning is consistent across different but overlapping immediate neighborhoods and is stable within an immediate neighborhood.

With this simple hashing-based assignment, any nodes in YAPPERS can insert and delete $\langle key, value\rangle$ pairs into the system. For example, suppose node X wishes to insert a pair $\langle k, v\rangle$. Then for each node $Y \in IN(X)$, X locally computes whether $HASH(k) \equiv (HASH(IP_Y) \bmod b)$. If Y has the same color as the key k, X then sends a message to Y for storing the pair $\langle k, v\rangle$. Upon receiving such a message, Y is required to store the pair. Similarly, a lookup request for a key k is sent directly to a node responsible for k.

When a node X executes the hashing-based assignment described above, there are two potential pitfalls:

1) multiple nodes in $IN(X)$ have the same color as the key k
2) no nodes in $IN(X)$ have the same color as the key k.

We avoid pitfall 1 by allowing X to pick any one of these nodes to store the key. We avoid pitfall 2 through a *backup assignment* scheme. Specifically,

- (*Backup*): When there are no nodes in $IN(X)$ that have color C_i, color C_i is assigned to a node with color $C_{(i+1) \bmod b}$. If there are multiple nodes of color $C_{(i+1) \bmod b}$, choose the node with the smallest IP address.

For example, if a key k hashes to color C_5 and no IP addresses of the nodes in $IN(X)$ hash to color C_5, then k will be stored on a node with color C_6. If no such nodes exist as well, we try nodes with color C_7 and so on. This approach is similar to Chord's consistent hashing [10] except that we use b distinct hash values as opposed to all real numbers between 0 and 1.

To distinguish between the multiple colors of node X, we say a color C is the *primary color* of node X if X's IP address hashes to color C. Similarly, a color C' is a secondary color of node X if there is some node Y that *may* assign X, through the backup assignment, the color C'. Note that Y need not to have a key of color C' to store at X for X to have the secondary color C'. As long as Y could potentially have stored a C' key at X, then X must be searched when looking for C' keys, and hence X is considered to have the color C'. Thus by construction, every node has exactly one primary color and zero or more secondary colors.

In resolving the pitfalls mentioned above, our solution is no longer *consistent* and *stable* as envisioned earlier. For example, suppose node X is in two different immediate neighborhoods $IN(A)$ and $IN(B)$ and X's IP address hashes to color C_5. Suppose that in $IN(A)$, no nodes have color C_4. Then node A thinks X is responsible for both colors C_4 and C_5. However, node B might only know X is responsible for color C_5 assuming that another node in $IN(B)$ has color C_4, thus causing an inconsistency. Similarly, if a node Y with color C_4 joins A's immediate neighborhood, then we need to move all keys of color C_4 from X to Y, thus reducing stability.

Despite this setback, the limited size of the immediate neighborhood isolates the impact of inconsistency and instability in YAPPERS. In reality, inconsistency and instability only occur in poorly connected portions of the overlay network (e.g., a chain of nodes) where the immediate neighborhood is small. By probabilistic analysis [14], it can shown that if a node A has $b \log b$ nodes in $IN(A)$ where b is the number of hash buckets, then, with high probability, there exists a node of each color.

B. The Extended Neighborhood

Since YAPPERS only stores $\langle key, value\rangle$ pairs in the owner nodes' immediate neighborhoods, the answers for a key search are scattered throughout the overlay network in many different neighborhoods. Thus to support a *Total Lookup*, i.e., all answers must be retrieved for a lookup, YAPPERS must have a mechanism for searching through all the neighborhoods.

Obviously, flooding the overlay network like Gnutella is a solution. However, such flooding disturbs many nodes that do not actually have any answers for the search. To avoid these disturbances, a node A keeps track of a bigger neighborhood than its immediate neighborhood so that it can make bigger "jumps" and avoid flooding its direct neighbors. Call this bigger neighborhood the *extended neighborhood* and denote it by $EN(A)$ for node A.

Before defining $EN(A)$, we first define the *frontier* of node A, denoted by $F(A)$, as all nodes that are not in $IN(A)$ but are directly connected to a node in $IN(A)$. Formally, if $N(v)$ is the set of nodes directly connected to node v, then

- $F(A) = \bigcup_{v \in IN(A)} N(v) - IN(A)$

With the frontier, the extended neighborhood $EN(A)$ is then simply the union of the immediate neighborhoods of all nodes in the frontier of A. Formally,

- $EN(A) = \bigcup_{v \in F(A)} IN(v)$

Figure 2 illustrates the relationships between the immediate neighborhood $IN(A)$, the frontier $F(A)$, and the extended neighborhood $EN(A)$. In this figure, $h = 2$. So nodes B and C are in $IN(A)$. Nodes D and E are in the $F(A)$ because they are connected to B and C respectively. Therefore, $EN(A)$ includes H (part of $IN(D)$) and J (part of $IN(E)$).

Because YAPPERS defines $IN(A)$ as all nodes within h hops of A, the above definition of $EN(A)$ means that the extended neighborhood of A consists of all nodes within $2h+1$ hops of A. Using this extended neighborhood, we now describe the protocol for searching through all the neighborhoods, and provide a proof that guarantees all nodes of a given color are searched.

1) Searching with the Extended Neighborhood: Informally, when a node A wants to look up a key k of color $C(k)$, it picks a node with color $C(k)$ in $IN(A)$. If there are multiple such nodes, pick one at random. So say node B has color $C(k)$ in $IN(A)$. Node A then tags the lookup request with a unique identifier and sends the request to node B.

Node B, upon receiving the query, returns its local answers to A. Afterwards, node B determines which nodes are in its frontier $F(B)$. The frontier nodes are important because they *do not* store the key k at B. Hence by finding out where its frontier nodes store the key k, node B can find other nodes of color $C(k)$. Moreover, the computation of finding other nodes of color $C(k)$ can be done locally at node B without any communication because $EN(B)$, by construction, contains the immediate neighborhoods of all the frontier nodes. Therefore, we can flood only a subset of the nodes and not disturb any nodes that could not have any answers. To avoid cycles in the

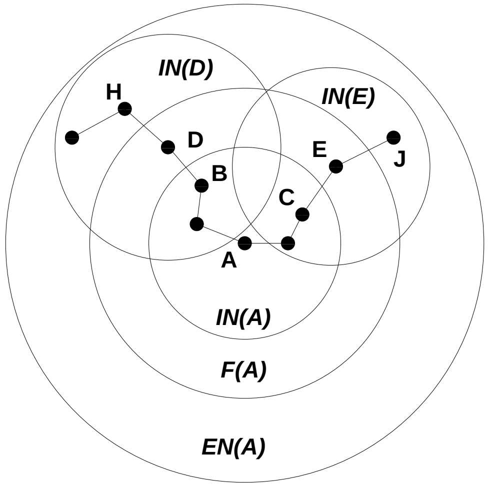

Fig. 2. The extended neighborhood of node A, $EN(A)$, is the union of the immediate neighborhood of nodes in the frontier of A.

forwarding step, the unique identifier, provided by the source of the lookup request, is cached and used to break the cycles.

The example in Figure 1 in Section III (where $h = 1$) illustrates this search pattern. When node A initiates the search, node A finds that B is colored gray. Thus A contacts B. Now B returns its local content and examines node D—its frontier node that is 2 hops away. Since E is the only gray node in $IN(D)$, node B forwards the request to node E. Node E forwards the request to F. Node F, after examining at its frontier node H, reaches I.

Notice that when a node X determines another node Y has the same color, we allow X to contact Y directly to forward a request, even if X and Y are *not* neighbors in the original overlay. So we are augmenting the overlay with additional connections. However, these connections respect the original overlay. Moreover, we do not create a new overlay by adding these forwarding connections between nodes of the same color, hence they cannot be used for any other purposes such as constructing immediate and extended neighborhoods. This approach is different from Chord or CAN where connections between nodes can be created at random and become part of a new overlay.

The pseudocode for performing total lookup, described informally above, is given in Figure 3. The lookup routine uses two helper functions *select* and $forward$ to determine which nodes have a specific color C and which nodes to forward the request to respectively. To do partial lookups, we can modify $total\_lookup$ to include a hop count limit or use random walks. For completeness, we also include the pseudocode for inserting a $\langle key, value\rangle$ pair. Deleting a $\langle key, value\rangle$ pair is similar to inserting one.

2) Completeness: The $total\_lookup$ algorithm guarantees that a search for a key eventually reaches all nodes storing the key. In other words, starting from any node of color C, we can reach all other nodes of color C using only the $forward$ routine described in Figure 3. Stated formally,

Theorem 1: (Completeness) For any two nodes A and B of color C, there exists a sequence of nodes $A = Z_0, Z_1, Z_2, \ldots, Z_{w-1}, Z_w = B$ such that for all $i < w$, Z_i has color C and Z_i forwards the request to Z_{i+1} when executing the $forward$ routine.

```
total_lookup(x, k):
  // find nodes that may have key k
  S = select(HASH(k), IN(x));
  choose a random Y in S;
  // start the search
  result = forward(Y, k, unique_tag());
  return result;
end

forward(x, k, tag):
  if tag not in cache then // check cycle
    answer = local_lookup(k);
    cache += tag;
    // find other nodes with key k
    for each node v in (IN(x) + F(x))
      S = select(HASH(k), IN(v));
      // forward request
      for each node w in S
        answer += forward(w, k, tag);
      endfor
    endfor
    return answer;
  endif
end

// find the subset of nodes in S
// that has color Ck
select(Ck, S):
  retval = {};
  // find nodes with hash value Ck
  for each v in S
    if HASH(v) = Ck mod b then
      retval += v;
    endif
  endfor
  if retval == {} then // if no nodes,
    // resort to the backup scheme
    backup = select(Ck+1, S);
    // however, only need one backup
    retval = Y in backup with smallest IP;
  endif

  return retval;
end

insert(x, k, v):
  // find nodes that can store <k,v>
  S = select(HASH(k), IN(x));
  choose a random Y in S;
  store(Y, k, v); // tell Y to store <k, v>
end
```

Fig. 3. Pseudo code for $total\_lookup$ and $insert$ in YAPPERS. Both procedure uses the helper functions $select$ and $forward$.

Proof: Without loss of generality, suppose there are two colors, white and black. We prove by contradiction. Suppose our claim is false. Then, there exist pairs of black nodes X and Y where we cannot go from X to Y by following a sequence of black nodes using the $forward$ routine. Since there are a finite number of such pairs of black nodes, we pick a pair of nodes A and B such that the distance between A and B in the overlay is the smallest among all pairs of nodes that do not satisfy our claim.

Now consider the shortest overlay network path $p_0, p_1, \ldots, p_s$ from A to B where $p_0 = A$ and $p_s = B$. There are two cases: $s \geq h + 1$ and $s < h + 1$. We show that both cases lead to contradictions, hence prove our claim.

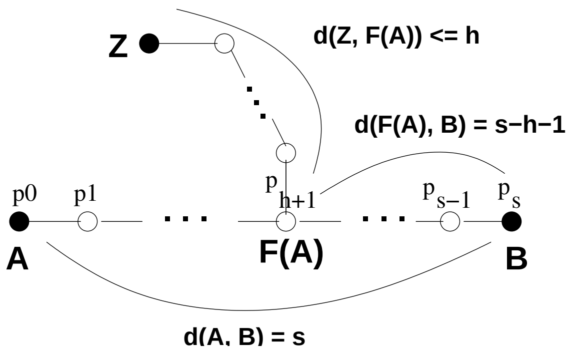

Fig. 4. Assuming two black nodes A and B are the closest nodes that cannot reach each other and is at least $h+1$ hops away, we can construct another pair Z and B where Z is even closer to B and cannot reach B.

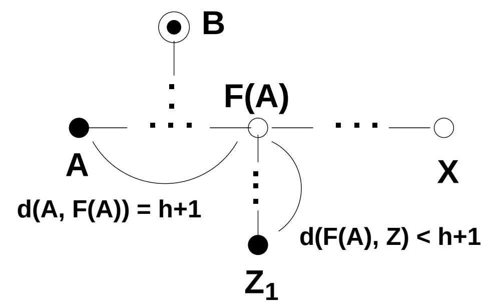

Fig. 5. Node X assigns the color black to node B using the backup mechanism, thus causes a pair of black nodes (A and B) to be within h hops while not knowing about each other.

Case 1: $s \geq h+1$. This case corresponds to the scenario shown in Figure 4. Consider the frontier node $F(A) = p_{h+1}$. There exists a black node $Z \in IN(F(A))$ (through the backup assignment in the worst case). Since Z is at most h hops away from $F(A)$, $Z \neq A$ because $s \geq h+1$. Also, there does not exist a sequence of black nodes from Z to B using the $forward$ routine because otherwise we can construct a sequence from A to B going through Z, which is a contradiction.

Now consider $\delta(Z, B)$, the distance between Z and B. By the triangle inequality, $\delta(Z, B) \leq \delta(Z, F(A)) + \delta(F(A), B)$. By construction, we know $\delta(Z, F(A)) \leq h$ and $\delta(F(A), B) = s - h - 1$. Therefore, $\delta(Z, B) \leq h + s - h - 1 = s - 1$, which means Z is closer to B than A and is a contradiction to our choice that A and B are the closest pair of black nodes that do not satisfy our claim. Hence, the case $s \geq h+1$ cannot happen.

Case 2: $s < h+1$. First note that this case can only occur if node B has multiple colors and its primary color (based on the IP address) is not black. Otherwise, node A would forward the request directly to node B when checking its own immediate neighborhood $IN(A)$. Figure 5 captures the scenarios for this case where node X is the culprit that assigned the extra black color to node B. As the figure depicts, node B does not have to be on the overlay path from A to X.

To prove that case 2 is also a contradiction, we explicitly construct a sequence of black nodes that allows node A to reach B. Let $t = \delta(A, X)$, the distance between A and X. If $t \leq h+1$, then $X \in IN(A) \cup F(A)$ and node A would have learned that node B is also black when determining where X stores black keys. Thus, node A will directly reach B, a contradiction. So it must be that $t > h+1$. In this case, consider the frontier node $F(A)$ in Figure 5. If $F(A)$ has assigned the color black to B, then again we have a contradiction. (When A examines its frontier nodes, it would discover that B is black, and again, A would directly reach B.) So we assume that $F(A)$ has not assigned B the color black. By the backup assignment scheme, $F(A)$ must know of at least one black node in $IN(F(A))$, call this node $Z_1 (\neq B)$. Note that $\delta(F(A), Z_1) \leq h$, and node A will reach Z_1.

For the same reason as $t > h+1$, $\delta(Z_1, X) > h+1$. However, $\delta(Z_1, X) \leq \delta(Z_1, F(A)) + \delta(F(A), X) \leq h + (t - h - 1) = t - 1 < t = \delta(A, X)$. Therefore if we repeat the step above (with A replaced by Z_1), we can find Z_2 such that Z_1 forwards the request to Z_2 and $\delta(Z_2, X) \leq t - 2$. Since each step brings us at least one node closer to X, eventually, in a finite number of steps $i \leq t - h - 1$, we get $\delta(Z_i, X) \leq h+1$. When this happens, $X \in IN(Z_i) \cup F(Z_i)$ which means node Z_i can infer node B is a black node and forward B the lookup request. Therefore, we can forward the request from A to B via the sequence $A, Z_1, Z_2, \ldots, Z_i, B$. This contradicts the assumption that there exists no sequence of black nodes between A and B using $forward$.

■

C. Maintaining Topology

So far we have assumed that each node in YAPPERS has enough local overlay-network topology information to determine its immediate and extended neighborhoods. We now focus on propagating topology changes as nodes enter and leave the overlay. We first discuss edge deletion and insertion and then proceed to node departure and arrival.

1) Edge Deletion: When an edge (X, Y) is deleted from the topology, distances between some nodes might increase which, in turn, may cause some YAPPERS nodes to shrink their immediate and extended neighborhoods. This behavior implies that we can limit the propagation of an edge deletion event to nodes that have both X and Y in their extended neighborhoods. Since YAPPERS uses a $2h+1$ hops extended neighborhood, an edge deletion requires both X and Y to broadcast the deletion event to its surviving neighbors with a time-to-live field of $2h$ hops.

Upon receiving the broadcast, each YAPPERS node updates the topology and adjusts its immediate and extended neighborhood accordingly. Note that changes in extended neighborhood has no effect on the node other than in determining future query-routing decisions. However, if the immediate neighborhood changes, the affected node may have to re-add some $\langle key, value \rangle$ pairs. For example, suppose node X is no longer in node A's immediate neighborhood after the edge is deleted, then A will have to find a node in the new immediate neighborhood to re-add A's $\langle key, value \rangle$ pairs that were previously stored on X.

2) Edge Insertion: Unlike edge deletions, broadcasting a new edge's presence is not sufficient for YAPPERS nodes to maintain the topology. Consider the case where two nodes X and Y were previously unknown to each other. When the new edge (X, Y) is added, X needs to know about Y's neighbors to rebuild its immediate and extended neighborhood. Moreover, nodes previously connected to X may also needs to know about Y's neighbors.

The naive solution is to do the same thing as edge deletion but append both X's and Y's new extended neighborhood information in the broadcast. However, note that if Z was previously connected to X, Z only needs the topology information that is within $2h-1$ hops of Y because Z only cares about nodes that are $2h+1$ hops away and these nodes are at most $2h-1$ hops away from Y. Similarly, if node W was previously connected to Z, then W only needs topology information for nodes that are $2h-2$ hops away from Y. Using this observation, each YAPPERS node can "trim" the topology information appended to the edge insertion broadcast and pass along only useful topology to downstream nodes. Similar to edge deletion, if the immediate neighborhood changes, then data relocation might be necessary if secondary colors can be moved to the newly added nodes.

3) Node Departure and Arrival: When a node X with w edges leaves the network, we treat the departure as w edge deletions. Each of X's neighbors is responsible for initiating a broadcast for the appropriate edge. As a side benefit of this approach, we do not require node X to depart gracefully.

A node arrival is only slightly more involved. As node X appears on the network, it first asks its new neighbors for their current views of the topology. Node X then merges these views to create its own extended neighborhood. Afterwards, node X initiates an edge insertion broadcast to each of its new neighbors appended with the appropriately trimmed subset of the new topology.

Since both node arrival and departure only affect other nodes within $2h$ hops and is independent of the rest of overlay, YAPPERS should scale better and be more stable than systems such as Chord that support one overarching hash table. For instance, in Chord, multiple node arrivals or departures will interact with each other and cause complicated reorganization whereas YAPPERS isolates each arrival or departure to a small neighborhood.

D. Summary

In short, YAPPERS builds a hybrid network that retains the advantages of both unstructured P2P networks and structured DHT networks. Specifically, within an immediate neighborhood, YAPPERS behaves like a DHT where pinpoint lookup queries are very efficient. When using extended neighborhoods to navigate between nodes of the same color, YAPPERS acts like Gnutella but with more intelligence. Notice that, unlike pure DHT-based systems, all nodes in YAPPERS participate to resolve searches even if there are more nodes than keys, which means hot spots (where many requests go to one node) are less common in YAPPERS.

V. ENHANCEMENTS

In experiments we performed to evaluate our basic YAPPERS design (see Section VI), we observed some performance shortcomings when running YAPPERS on networks with highly variable node degrees (e.g., a power-law type overlay). Specifically, we noticed two problems: the *fringe node* problem and the *large fan-out* problem.

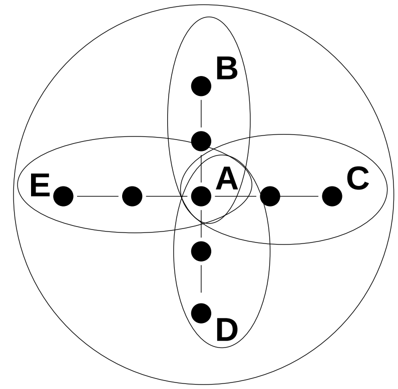

Fig. 6. In a Star topology, the central node A is overloaded by the fringe nodes B, C, D, and E as they assign large chunks of key space to A.

In the *fringe node problem*, a low connectivity node, through the use of backup assignment, allocates a large number of secondary colors to its high-connectivity neighbor which has no desire for the extra colors. Consider the star example in Figure 6. Suppose the keyspace is divided into 36 colors. Then the central node A, having 9 nodes in its immediate neighborhood, expects to handle its own primary color and three secondary colors, for $\frac{1}{9}$th of the total colors. However, the four fringe nodes B, C, D, and E will each assign 11 secondary colors to node A because their immediate neighborhood only has three nodes. As a result, node A is routing lookup requests for 12 out of the 36 colors, or one third of the total colors, which is much larger than the expected $\frac{1}{9}$th of the total colors.

The *large fan-out* problem captures a different challenge when forwarding search requests to other neighborhoods. Recall that YAPPERS uses frontier nodes to decide where to forward the request. So when a node A does the forwarding, each of A's frontier nodes may point to one or more different forwarding nodes. Consequently, the forwarding fan-out degree at node A is proportional to the number of the frontier nodes. If an overlay network's average node degree is 5, then the fan-out degree is $O(5^{h+1})$ (which is 125 for $h = 2$). As we will discuss more in the evaluation section, this large fan-out is desirable when doing partial lookup because we can reach more nodes quickly. On the other hand, for total lookup, large fan-out can be undesirable due to duplicate requests when forwarding.

A. Prune Fringe Nodes

One obvious solution to the *fringe node* problem is to prune away low connectivity nodes. For example, we can recursively prune all nodes with degree 1 from the overlay network to get rid of leaf nodes. With a small risk of disconnecting the overlay, we can also prune away nodes with degree 2 to eliminate long chains of nodes connecting two large components.

To implement pruning in YAPPERS, a node X, upon arrival, determines whether it is a fringe node based on its local topology information. If X is a fringe node, then X does not participate in YAPPERS directly. Instead, node X selects a nearby high connectivity node Y as its proxy. So when X wants to register a $\langle key, value\rangle$ pair or do a lookup search, node X sends the request to node Y and asks Y to perform the task on its behalf.

The trade-off in using pruning is the extra workload on the proxy node Y, generated by the nearby fringe nodes. However, this extra workload is smaller than handling an extra color C and forwarding requests for color C in the entire overlay network. Note that this approach is similar to organizing a Gnutella network with super-peers. The distinction here is that super-peers, in general, are determined based on bandwidth constraints whereas connectivity is the criterion used in YAPPERS.

B. Biased Backup Node Assignment

An alternative solution to the *fringe node* problem is to bias the backup assignment scheme so that high connectivity nodes do not get extra colors. Formally,

- (*Bias*): Node X can assign a backup color to node Y if and only if $\alpha \cdot |IN(X)| > |IN(Y)|$, where α controls the relative sizes of the neighborhoods.

In other words, we forbid a node with a small immediate neighborhood assigning backup colors to a node with a large immediate neighborhood. In the worst case, if a node X is unable to assign any backup colors to any nodes in $IN(X)$, then node X manages these extra colors itself.

The drawback of using the bias is the increase in the number of nodes we must contact for a lookup. For the star example in Figure 6, suppose without the bias, all the fringe nodes assign a secondary color C to the central node A. Then a lookup for a key of color C will hit only node A. However, if we are using bias where $\alpha = 2$, then all four fringe nodes have to retain color C themselves. So a lookup for C will hit 4 nodes instead of 1.

C. Reducing Forward Fan-out

For the *large fan-out* problem, we can apply three steps to reduce the fan-out degree, thus reducing the number of duplicate messages in query broadcasts. Specifically, when node A forwards a lookup request for color C,

1) If a frontier node F assigns C to node B via the backup mechanism, then forward the request to B.
2) If a frontier node F assigns C to a set of nodes S, do not forward to any nodes in S if $S \cap IN(A) \neq \emptyset$. Otherwise, only forward to one of the nodes in S.
3) In step (2), when choosing one node from S, try to pick common nodes between multiple frontier nodes.

Step (1) is necessary because the only way of reaching a backup node B could be through the frontier node F. If no backups are necessary, then steps (2) and (3) try to avoid forwarding to far-away nodes if a closer one exists.

There are several other alternatives. For example, after node A has decided to forward the request to a set of nodes S, node A can include S in the forwarded message to help other nodes in reducing the fan-out. Another solution is to run an additional pass on the set S to see if two nodes $A, B \in S$ will forward the request to each other through some other path. If so, we pick only one node to forward the query to. Finally, we can move away from the idea of using query flooding for total lookups. For example, when a node desires to initiate a query, it contacts a fixed number of random nodes in the network and forwards the query to them. Each of these random nodes can then forward the query to their neighbors with a TTL of 1. By carefully controlling how many nodes are contacted in the first step, and how they are chosen, we can provide probabilistic guarantees on the total number of nodes contacted for the query.

VI. EVALUATION

To estimate statistics on YAPPERS running over a real overlay network, we simulated YAPPERS on a snapshot of the Gnutella network [15] containing $24,702$ connected nodes and several synthetically-generated regular graphs. In our YAPPERS implementation, we use $h = 2$. For $h > 2$, the extended neighborhood is too large because it includes all nodes within $2h + 1$ hops. On the other hand, for $h = 1$, the immediate neighborhood is too small to efficiently support more than 4 or 5 colors.

With our implementation, we examine the search efficiency with respect to the expected fraction of nodes contacted per query, the search overhead in terms of the fan-out degree for forwarding search messages, and the optimal b (number of hash buckets) to use. We focus our discussion of the results on the Gnutella snapshot and will mention relevant points about regular graphs when appropriate.

In our experiments, we compare YAPPERS to the simple Gnutella protocol. In reality, most Gnutella-based systems use a 2-tier architecture, with low-bandwidth clients maintaining a single connection to a *supernode*, and the supernodes themselves forming a standard Gnutella network. We note that we can adapt YAPPERS to this architecture by involving only the supernodes in YAPPERS.

A. Search Efficiency

The efficiency of executing a total-lookup request is captured in the expected fraction of nodes contacted, denoted by $\bar{F}$, during the search. This fraction $\bar{F}$ is equal to $\frac{\bar{C}}{b}$, where $\bar{C}$ is the average number of colors assigned to each node and b is the number of hash buckets (colors) used. To see this, notice that $N \cdot \bar{C} = (N \cdot \bar{F}) \cdot b$ because $N \cdot \bar{C}$ is the total number of colors in the system, $(N \cdot \bar{F})$ counts the number of nodes having a particular color, and hence $(N \cdot \bar{F}) \cdot b$ also counts the number of colors in the system.

Table I illustrates how different enhancements of YAPPERS affect $\bar{C}$ and $\bar{F}$ when running over the Gnutella snapshot with $b = 32$. The last line in the table provides the baseline comparison of running straight Gnutella. From the table, notice that to execute a total lookup, Gnutella has to contact every node. In contrast, YAPPERS only contacts 8% to 18% of the nodes depending on the enhancement.

Enhancements	Nodes in Overlay	Avg Colors per Node ($\bar{C}$)	Avg Nodes Contacted per Query ($\bar{F}$)
None	24,702	3.73	11.6%
Pruning (Degree 1)	15,785	3.10	9.6%
Pruning (Degree 2)	12,081	2.64	8.2%
Bias ($\alpha = 2$)	24,702	5.90	18.5%
Gnutella	24,702	(N/A)	100%

TABLE I

SEARCH EFFICIENCY OF RUNNING YAPPERS WITH $b = 32$ ON GNUTELLA SNAPSHOT AND VARIOUS ENHANCEMENTS

More specifically, Table I show that as we prune away more fringe nodes, $\bar{F}$ decreases from 11.6% to 9.6% and then to 8.2%. Accordingly, $\bar{C}$ also decreases from 3.73 to 2.64. This observation correlates with our intuition of the *fringe node* problem (described in Section V) where fringe nodes are assigning extra secondary colors to highly connected nodes. Also as expected, using the bias enhancement increases $\bar{C}$ since a color previously assigned to a highly connected node has been moved to many fringe nodes. Consequently, a larger fraction of the nodes are contacted during a lookup. Despite this increase, in the next section we show biasing actually incurs less overhead in terms of messages generated when a search must reach every node in the overlay.

For completeness, we show the cumulative distribution of colors per node for YAPPERS in Figure 7. The x-axis shows the number of colors assigned to a node and the y-axis shows the percentage of nodes with equal or fewer colors. From the figure, we note that there is a small fraction of nodes with a large number of colors. Upon close examination of these nodes, we found that without using bias, this small fraction is composed of high-degree nodes. With biasing, this fraction consists of leaf nodes that are unable to give away extra colors due to the bias constraint. Figure 7 also provides further evidence that pruning indeed reduces the number of colors on the high degree nodes as we reach the 100 percentile at 11 colors per node with pruning as opposed to 27 without.

For a regular graph with the same number of nodes and edges, we found similar savings in the number of nodes contacted. However, the distribution of colors per node is not heavy-tailed in that only 1.1% of the nodes have more than 8 colors and the maximum is 13 colors per node. Also, applying pruning or biasing has no impact since there are no fringe nodes in a regular graph.

B. Search Overhead

As we have seen, YAPPERS is more efficient for searching than Gnutella in the sense that a lookup is processed by fewer nodes, specifically, an order of magnitude fewer nodes than Gnutella. However, YAPPERS propagates a lookup much more "aggressively" through the original overlay. In particular, when a YAPPERS node forwards a lookup to nodes in other neighborhoods, the fan-out degree is large due to the large number of frontier nodes. To capture this fan-out degree, for each node X and color C, we determine the fan-out degree in the number of nodes X would forward the request to when looking up a key of color C. We then average over all colors and all nodes.

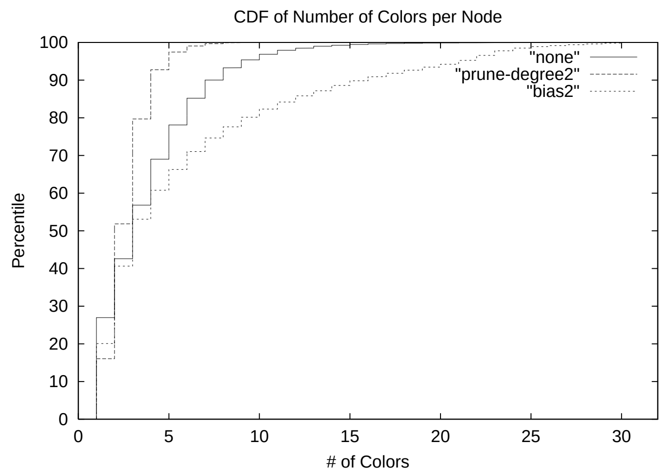

Fig. 7. Cumulative distribution plot of number of buckets per node for YAPPERS with $b = 32$ and no enhancements running on Gnutella snapshot.

Enhancements	Avg Fan-out Degree
None	835.3
ReduceFanOut	140.8
Pruning (Deg. 2), ReduceFanOut	160.9
Bias ($\alpha = 2$), ReduceFanOut	82.5
Gnutella	5.2

TABLE II

THE FANOUT DEGREE WHEN RUNNING YAPPERS WITH $b = 16$ ON GNUTELLA SNAPSHOT AND VARIOUS ENHANCEMENTS

Table II shows the average fan-out degree for YAPPERS with various enhancements. Notice that the basic version has a high fan-out degree of 835. As we apply the fan-out reduction techniques and avoid overloading high connectivity nodes with extra colors, the fan-out is reduced to 82. For a regular graph of comparable size, the average fan-out degree is 62.

Large fan-out degree has both positive and negative impact on performance. On one hand, lookups are propagated much faster through the network in YAPPERS. On the other hand, we need extra state information to keep track of these nodes in the extended neighborhood and connection states (if any). Also, when flooding the entire overlay of a given color, many forward messages could potentially be duplicates.

Fortunately, for partial lookups, large fan-out is actually desirable. Consider an example where a lookup wants 50 values for a particular key. In Gnutella, that means flooding all nodes within 5 or 7 hops. In YAPPERS, such a lookup can be answered in one forwarding step to the 80 or so nodes because these 80 nodes contain data from nearly all nodes within 5

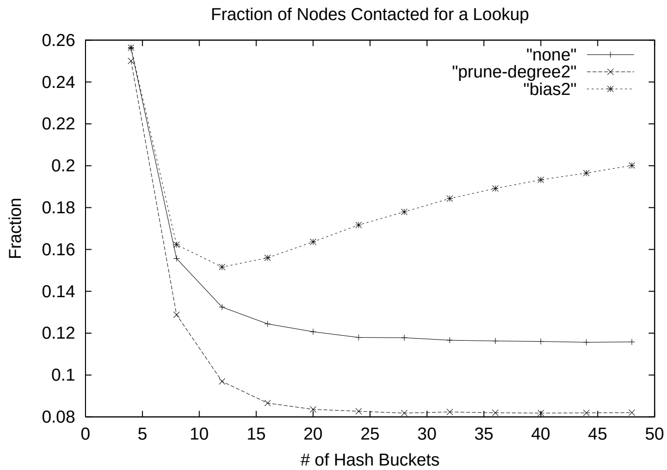

Fig. 8. Fraction of nodes contacted during a lookup

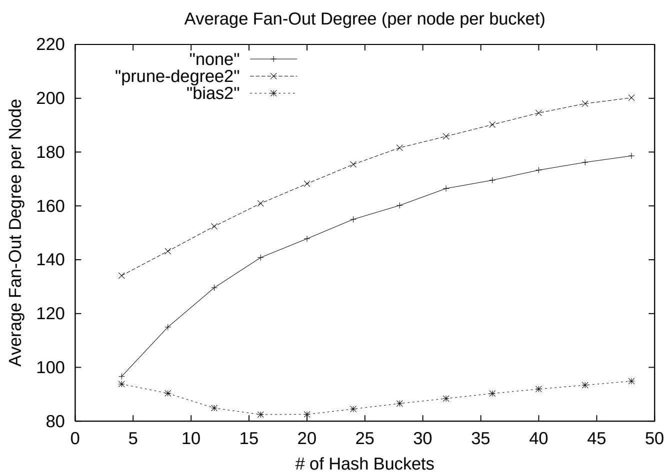

Fig. 9. Average fan-out degree per node on a lookup

hops. Thus we not only get the answer faster but also only contact nodes that have a much higher probability of having a result. However, if the lookup requires searching through the entire network, then the redundant forward messages, caused by the large fan-out, could be overwhelming. In those cases, random walk[7] or iterative deepening[6] techniques can help to reduce the redundant messages.

C. How Many Buckets to Use

The natural question is: how many hash buckets (colors) is ideal for YAPPERS? To answer this question, we ran YAPPERS with $b = 4, 8, \ldots, 48$. The results are shown in Figures 8 and 9.

First, Figure 8 shows the search efficiency where we have the fraction of nodes contacted during a lookup on the y-axis and the number of hash buckets, b, used on the x-axis. As we increase the number of buckets, the fraction of nodes contacted does not improve significantly for $b > 20$. In the case of using biasing, the best case is $b = 12$, and actually deteriorates for larger b. The reason for the lack of continuing improvement beyond $b = 20$ is that the size of the immediate neighborhood is the same regardless of how many buckets are used. Thus the best possible condition occurs when every node in the neighborhood is assigned exactly one bucket (color) and having more buckets (colors) than nodes does not help matters.

Second, Figure 9 shows the overhead in terms of the average forwarding fan-out degree per node per hash bucket (color). The y-axis shows the average degree and the x-axis shows the number of hash buckets, b, used. Unlike the search efficiency, the average fan-out degree increases with larger b. However, with bias, we get a relatively constant fan-out degree as the number of buckets change.

To balance the conflicting trend of the search efficiency and the search overhead, we see that $b = 16$ is the sweet spot for this Gnutella snapshot. If the number of buckets is smaller, then we are contacting more nodes than necessary (as seen in Figure 8). If the number of buckets is larger, the increase in fan-out degree (seen in Figure 9) may render the gain in contacting fewer nodes irrelevant. Of course, the best b depends on the underlying topology. For example, $b = 12$ is the best for a regular graph of comparable size.

Notice that even though we cannot increase b arbitrarily to achieve better performance, we can apply our algorithm recursively to all nodes responsible for one hash bucket. In other words, manage each of the b sub-networks with another YAPPERS network. With this recursion, we can increase our performance further.

VII. Conclusion and Future Work

In this paper, we proposed our YAPPERS scheme for building an efficient peer-to-peer search mechanism without explicit control of the overlay network. Specifically, our scheme is a hybrid that uses distributed hash tables (DHTs) in small areas and uses intelligent forwarding over large areas. The main advantages are that our scheme:

- disturbs only a small fraction of the nodes in the overlay for each search.
- does not require restructuring the underlying overlay network.
- each node requires only knowledge of a small neighborhood and is independent of the rest of the overlay, and is thus less affected by node arrivals and departures.

For future work, we want to better quantify YAPPERS' performance gains when doing partial lookups. In particular, what is the best strategy for forwarding a partial lookup through YAPPERS' large fan-out network? What are the expected savings of contacting specific nodes in nearby neighborhoods as opposed to Gnutella flooding all nodes within 5 or 7 hops? Besides evaluating partial lookups, we also want to study the effect of frequent node arrivals and departures on YAPPERS. Notably, how does YAPPERS' performance degrade in an unstable network compared to DHTs?

Acknowledgments

We thank Adam Meyerson for his inputs on approximation algorithms for the dominating set problem which led to the notion of the extended neighborhood. We also thank Evan Greenberg for discussions regarding maintaining the topology information.

REFERENCES

[1] "Gnutella," Website http://gnutella.wego.com.

[2] FastTrack, "Peer-to-peer technology company," Website http://www.fasttrack.nu/, 2001.

[3] G. Pandurangan, P. Raghavan, and E. Upfal, "Building low-diameter peer-to-peer networks," in *Proceedings of 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, 2001.

[4] E. Cohen, H. Kaplan, and A. Fiat, "Associative search in peer to peer networks: Harnessing latent semantics," Stanford Networking Seminar.

[5] "Morpheus," Website http://www.musiccity.com.

[6] B. Yang and H. Garcia-Molina, "Efficient search in peer-to-peer networks," in *Proceedings of the 22nd IEEE International Conference on Distributred Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[7] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proceedings of the 16th annual ACM International Conference on Supercomputing (ICS)*, 2002.

[8] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *Proceedings of the 22nd IEEE International Conference on Distributred Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, San Diego, August 2001, pp. 149–160.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, San Diego, August 2001, pp. 160–177.

[11] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of SOSP '01*, 2001.

[12] B. Y. Zhao, J. Kubiatowicz, and A. Joseph, "An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, 2001.

[13] P. Ganesan, Q. Sun, and H. Garcia-Molina, "A case for locally-organized peer-to-peer lookup service," Tech. Rep., Stanford University, 2002, Available at http://dbpubs.stanford.edu/pub/2002-60.

[14] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[15] Clip2.com, "Clip2 gnutella crawl files," Private collection.