Speculative Routing and Update Propagation: A *Kundali* Centric Approach

Aditya Mohan[1], Vana Kalogeraki
Intelligent Enterprise Systems Laboratory
HP Laboratories Palo Alto
HPL-2002-237
August 22nd , 2002*

E-mail: a@cs.brown.edu, vana@hpl.hpl.com

routing, update propagation, bloom filters, peer-to-peer

Peer-to-peer networks have gained much attention due to their attractive features of self-organization, scalability and decentralized control. The key challenge in these networks is how to efficiently locate and retrieve the correct data. Techniques for efficient searching in peer-to-peer networks have been recently proposed; however, these handle location and routing as a single problem and impose a structure in the network by mapping the data to particular nodes. In this paper, we propose propagation and routing algorithms for a fully decentralized, self-organizing network. Our goal is to maximize the probability of finding the data, minimize peer access latencies and balance the workload among many peers. Central to our approach is the Kundali data structure that represents the set of data maintained by the peers and drives the smart routing of the search requests (queries). Kundali, for each peer, maintains a Bloom Filter based set of synopsis of the data expected to be present at each routing *direction*. Requests that cannot be answered locally, are propagated only to those *immediately* connect peers whose synopsis depict the closest match. We have implemented our algorithms in the context of a fully decentralized Internet caching service in our internal HP network. Our mechanism is inexpensive, highly scalable, resilient to node failures and with no administration cost. Experimental results validate our algorithms and show that they have good performance results.

* Internal Accession Date Only Approved for External Publication
[1] Department of Computer Science, Brown University, RI, 02912
© Copyright Hewlett-Packard Company 2002

Speculative Routing and Update Propagation: A *Kundali* Centric Approach

Aditya Mohan
Department of Computer Science, Brown University, RI, USA 02912
a@cs.brown.edu

Vana Kalogeraki
HP Labs, 1510 Page Mill Road, Palo Alto, CA 94304
vana@hpl.hp.com

***Abstract* - Peer-to-peer networks have gained much attention due to their attractive features of self-organization, scalability and decentralized control. The key challenge in these networks is how to efficiently locate and retrieve the correct data. Techniques for efficient searching in peer-to-peer networks have been recently proposed; however, these handle location and routing as a single problem and impose a structure in the network by mapping the data to particular nodes. In this paper, we propose propagation and routing algorithms for a fully decentralized, self-organizing network. Our goal is to maximize the probability of finding the data, minimize peer access latencies and balance the workload among many peers. Central to our approach is the Kundali data structure that represents the set of data maintained by the peers and drives the smart routing of the search requests (queries). Kundali, for each peer, maintains a Bloom Filter based set of synopsis of the data expected to be present at each routing *direction*. Requests that cannot be answered locally, are propagated only to those *immediately* connect peers whose synopsis depict the closest match. We have implemented our algorithms in the context of a fully decentralized Internet caching service in our internal HP network. Our mechanism is inexpensive, highly scalable, resilient to node failures and with no administration cost. Experimental results validate our algorithms and show that they have good performance results.**

Keywords: peer-to-peer networks, self-organizing networks, routing, update propagation, bloom filters.

I. INTRODUCTION

As computers become more pervasive and communication technologies advance, a new generation of peer-to-peer (P2P) networks are increasingly becoming popular for real-time communication, ad-hoc collaboration and resource sharing in a large-scale distributed environment. P2P networks create virtual (logical) networks with their own location and routing mechanisms that allow individual computers to share data (e.g., files) and resources (CPU, storage, etc.) directly, without the need for dedicated servers.

The topology of these networks and the routing mechanisms used, have been the focus of much research recently [14,17]. The challenge is, given the exact reference of an object of interest, how can you efficiently retrieve the correct object from a fully decentralized, self-organizing network? The location, intermittent connectivity, autonomy, and organization of the peers have a significant impact on the scalability, efficiency and performance of the distributed applications.

The importance of these problems has been recognized by recent P2P systems such as Oceanstore[8,11], Pastry[12], and Chord[13] that employ distribution and lookup protocols dictated by a consistent mapping between an object key and a hosting node. Location and routing is handled as a single problem, which is typically restricted by the frequent node arrivals and departures in the network. Early peer-to-peer systems such as Napster [15] and Morpheus [16] use centralized indexing server for the contents of the peers. Each peer that joins the network uploads the list of its files to the central server. Freenet [6] and Gnutella [7] on the other hand, are completely self-organizing; a node joins the network of peers by establishing an arbitrary and ad-hoc connection with at least one node in the network. Search is performed independently of the search query or the peers that can answer it. As a result, the messages travel a large number of hops from one node to another until the results are found. As measured in [2], the amount of bandwidth incurred by relaying transmission of 83[1] bytes in a network with an average of 8 connections per peer and where each message is propagated a maximum of 8 times, is 1,275,942,400 bytes.

Our approach builds upon the notion of immutable data objects [5] to produce references that uniquely characterize the data objects in the network. The distribution of the objects is driven by decisions made by the individual peers based only on the stored information available locally. Central to our approach is the smart routing of the queries and the dissemination of the updates. Each node creates and maintains a set of data synopsis for each of the direction where it can forward a query. This set of data synopsis, which we are calling *Kundali*, is the key component of our system. A data synopsis represents a summary of the data objects maintained by that peer. When the request cannot be served locally at the node, the node compares it with the summaries of its peers and sends the query message only to those peers

[1] IP header = 20 bytes, TCP header = 20 bytes, Gnutella header = 23 bytes, Minimum speed 1 byte, Search string 19 bytes (typical load)

whose summaries show promising results. Individual object operations (e.g., object create, object delete) are grouped into *group_update* operations and the incremental updates are propagated to the peers. Under our approach, the memory requirements as well as the processing and network bandwidth for routing and update propagation are significantly reduced.

Our mechanism has the following advantages:

- Maximizes the probability of finding the data as it routes the request messages (search requests) directly to those nodes that show promising results.

- Reduces the number of messages in the network and also the number of peers that process and propagate the messages.

- Balances the load in the network and reduces query latency by distributing the requests over multiple peers.

- Scales well with respect to the number of peers as it generates synopsis of data and periodically disseminates them to the peers.

Our routing and update propagation algorithms using Kundali, are used to build a fully decentralized Internet caching service. The cache is organized as a network of peers, each maintaining a set of files. There are no centralized servers or dedicated hardware to hold the files. The files are kept at the individual user's machines. Peers export their local caches to other nodes and cooperate to provide a wide-area caching functionality. Our mechanism guarantees that the requests are routed efficiently to the correct peers with the advantages of being inexpensive, highly scalable, resilient to node failures and requiring little administration.

The rest of the paper is organized as follows. Section 2 presents an overview of our system and Section 3 describes the Bloom Filter mechanism. In Sections 4, 5 & 6 we propose our routing and update propagation algorithms and in Section 7 we present our experimental results. Section 8 concludes the paper.

II. SYSTEM MODEL

We assume a logical network of nodes (peers) in which each node maintains connections with other nodes. The number of connections is typically limited by resources at the peer and is updated dynamically based on the peers' interests.

The network is self-organizing; peers use their own incentive-based policies to decide to which peers to connect to or to query in the system. A node searches in the network by sending request (Query) messages to its peers. The Query message contains a constraint that will be evaluated locally in each node to determine what results to return. When a node discovers that a peer frequently produces good results to its requests, it attempts to move closer to it in the network by connecting directly to that peer. A more detailed description of our self-organization algorithm is presented in [10].

Each peer is associated with a local store for the repository of the data objects (files). Files are uniquely identified by the means of intrinsic references [5], which are generated when the files are inserted in the system. Intrinsic references are based on the hash digest of the actual contents of the files rather than their name or location and allow us to create persistent, state-independent and immutable storage. Each file is characterized by meta-data such as title, fetch date, expiry date, etc. When a new file is inserted or existing files are updated or deleted from the repository, the node also updates the corresponding meta-data.

Each node builds a synopsis of the files in its store and sends it to its direct peers. The node uses its peers' synopsis to decide where to route the request next. When a search request arrives at the node, the node evaluates the request locally (searches through its local store) and if the request cannot be served locally, it then routes the request only to those of each peers whose synopsis have the closest match. The synopses are produced using the Bloom Filter approach, as described in the next section.

A. Messages in the Peer to Peer Network

The peers in the network communicate by exchanging the following messages:

- Ping/pong: A node sends a ping message to connect to the network of peers. A peer that accepts the connection replies with a pong message that includes the IP address of the sender node.
- Query: This is the primary mechanism for searching in the network and includes a constraint that will carry the search operation. If the peer node has a reply, it responds with a QueryHit message.
- QueryHit: Used as a reply to the Query message and includes information so that the recipient can acquire the corresponding data.
- Update: A node generates a new data synopsis when files are added, deleted or updated in the local repository, or when new peers join the network. It then uses the Update message to send the synopsis to its direct peers.

III. BLOOM FILTER BACKGROUND

Bloom Filters [1] are memory efficient randomized data structures whose functionality is to test the cardinality of a member in a group. The Bloom Filter data structure gives a

compact representation of the group by using an array of bits; each takes a binary one or a zero value. The cardinality of a contender is checked by comparing the bit array generated by hashing the contender by multiple hash functions, to the bit array of the Bloom Filter. This with certain errors gives a probabilistic answer to whether the contender is in the group or not. Even if the bits generated by hashing the contender has all the corresponding bits set in the Bloom Filter, there is a non-zero probability that the element may not be in the group. This is referred to as *false positives*. On the other hand if there is a bit in the contender's hashed bits that is not set in the Bloom Filter, we can certainly say that the contender element is not there in the group. Hence there is a non-zero probability of a false positive but a zero probability of a false negative. This fact makes the Bloom Filter approach highly suitable for a wide variety of distributed network applications.

A. Mathematics

The reduction in memory requirement for representing the membership information in a group is accomplished by exploiting the possibility that a small number of false positives may not greatly affect the system performance. There is a direct relation between the probability of a false positive and the number of bits one use for representation of the Bloom Filter.

Assume a group of n elements given by the set $S = \{a_1, a_2, ..., a_n\}$. The Bloom filter that represents the set S is described by a bit array *BF* of length *m,* all initially set to *0*. We assume k hash functions, $h_1, h_2, ..., h_k$ with $h_i : X \to \{1..m\}$. Each hash function maps each element of the set S to a value between $\{1..m\}$ in a totally random fashion. For each element $s \in S$, the bits at position $h_1(s), h_2(s), ...h_k(s)$ are set to 1. Note though, that, a bit may be set to 1 multiple times. To determine whether a certain element x is in S, we check whether all the bits given by $h_1(x), h_2(x), ...h_k(x)$ are set to 1. If any of them is 0, then we are certain that the element x is not in the set S. If all $h_1(x), h_2(x), ...h_k(x)$ are set to 1, we conclude that x is in S, although there is a certain probability that we are wrong (this is the case that a Bloom Filter may yield a *false positive).*

After inserting n elements into a Bloom filter of size m using k hash functions, the probability p_0, that a specific bit is still 0 is given by:

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}} \quad (1)$$

Hence, the probability of a false positive, that is, the probability that all k bits have been previously set is:

$$p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (2)$$

Given m and n, our goal is to optimize the number of hash functions k to minimize the false positive probability p_{err}. From (2), we derive that p_{err} is minimized for $k = \frac{m}{n} \ln 2$. This is also a global minimum. Thus for an optimal solution:

$$(1/2)^k = (0.6185)^{\frac{m}{n}} \quad (3)$$

In practice, to reduce the computational complexity, a small value for k is preferred.

B. Economics - Memory, Computational Complexity and Bandwidth

As shown in formula (3), there is a trade off between the number of hash functions to use (k), the size of the bloom filter (m) and the number of elements (n) in the network. Hence the performance of the Bloom Filter mechanism in a network is affected by the following factors:

- *The computational overhead to lookup a value* - number of hash functions to use
- *The memory requirement* - Bloom Filter size
- *The bandwidth requirement* - Bloom Filter size
- *Error rate* - false positives

Among these, the performance can greatly degrade if the error rate is high. This is the case where a query is propagated among peers that do not have the object in their repositories. To get an optimal solution, k and m need to satisfy formula (3). Furthermore, to reduce the computational complexity, k needs to be small. These force us to make the Bloom Filter size (m) large. This to some extent can makes things difficult for a peer, which may just be a PDA with 16MB of memory. But we believe that this is a reasonable requirement. For example, maintaining a Bloom filter of 128 bits size for each neighboring peer and assuming an average number of 5 connections per peer, the total memory requirement comes out to be 80 bytes, which is quite reasonable even for machines with strict resource requirements.

Another important consideration is the bandwidth consumption to propagate the bloom filter updates among the peers. [2] for example, compress the Bloom filter array before transmission by cleverly closing a combination of *k* and *m* so as to reduce the probability that a bit is 1 to 1/3, at the expense of introducing some extra computational complexity. To minimize the traffic in the network, our approach generates updates only when there is a change in the peer's local Bloom filters or when new peers join the network. The size of each Bloom filter message is just a few bytes. Furthermore, by grouping individual Bloom filter updates into *group_updates* we can further reduce the bandwidth consumption while maintaining reasonable performance. However, with the key provision that soon peers will be connected via low-latency, high-bandwidth networks, even peers running on wireless devices will be able to get a good quality of service.

Hence, in practice, with a reasonable size for the Bloom Filter *m* and a small number of hash functions *k*, the bandwidth consumption can be kept low while having minimal computational complexity.

IV. KUNDALI ARCHITECTURE

Our mechanism builds a novel data structure called Kundali, based on a modified version of the conventional Bloom Filter approach. Each peer maintains two sets of Bloom Filters; the Local Bloom Filter (*LBF*) that represents the objects (files) in the local repository, and Remote Bloom Filters (*RBF*) obtained from its *immediate*[2] peers. A Remote Bloom Filter is computed as the sum (OR) of the Bloom Filters maintained by that peer and therefore represents the view of that peer. Essentially, this gives us an indication of which data objects are reachable through that immediate peer. Hence, the set of local *Remote Bloom Filters* gives the approximate direction in which the queries should travel in order to maximize the probability of finding a particular object. By appropriately choosing the size of the Bloom Filter, we minimize the probability of false positives while maximizing the probability that the query makes its way to the exact location of the object.

Figure 1 illustrates the Kundali data structure maintained by each peer. It is important to note that the Remote Bloom Filter RBF(c) that node b maintains for c is not equal to node's c LBF(c). The reason is that, RBF(c) is essentially a summary of all the files that reside in node's c sub-network and computed as RBF(c) = LBF(b) + RBF(b) + RBF(d).

We represent a Local Bloom Filter by a 128-bit array where each bit is associated with a *counter*. The *counter* is responsible for maintaining the consistency of the Local

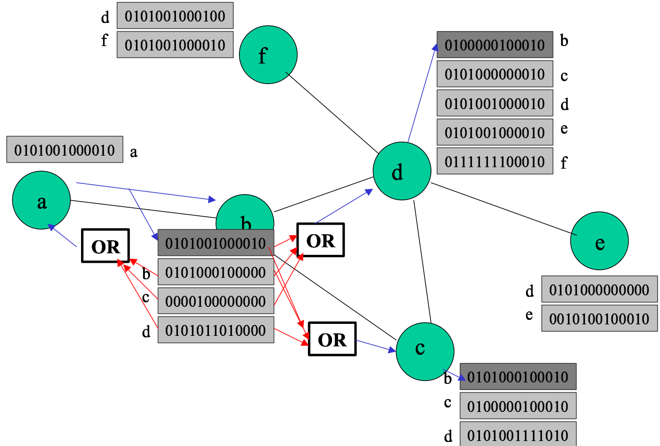

Fig. 1. Kundali update propagation when a new node ('a') joins the network

Bloom Filter when there is a change in the status of the local repository. When a file is added in the local repository, the corresponding bits in the filter are set to 1 and the associated counters are also initialized to 1. For each additional bit set in the Local Bloom Filter, the corresponding counter is incremented by 1. When a file is deleted, the corresponding counters are decremented. If a counter reaches zero, the corresponding bit in the Local Bloom Filter is reset to zero. This ensures that the Local Bloom Filter is able to give a reliable view of the local file repository.

V. ROUTING USING *KUNDALI*

This section describes our routing algorithm using the Kundali data structure.

A node searches in the network by sending Query messages to its peers. The Query message contains a constraint (the file's intrinsic reference and possibly some meta-data) that will be evaluated locally at each peer to decide what results to return. If the Query message cannot be served locally, the node has to decide to which of its peers to propagate the message next. The node sends the request to those peers whose Bloom Filters are among the k numerically closest to the object's id (intrinsic reference). The results are ranked based on the goodness of the comparison and the Query message is propagated only to a certain percentage of peers with highest ranks.

[2] Two peers are called immediate, if they share a direct open connection.

The rank is defined and updated dynamically by each peer. Hence, at each hop the Query is probabilistically routed closer to the correct destination. Essentially, if we propagate the message only to the highest ranked peer we perform a directed walk in the network. The advantage is that we reduce the resource consumption but at a much slower update rate, at the expense of potentially getting in-correct results. Our experimental results demonstrate that if we route the request to only one peer, we are almost always able to find an answer to our query. (For details, see Section 7).

When the file is found in the network, the Query message is no longer forwarded. To provide a termination condition so that messages are not propagated indefinitely in the network when no files are found, each message is associated with a *time_to_live* (TTL) field that represents the maximum number of times the message can be propagated in the network. The TTL value is decremented each time the message reaches a peer. A node that receives a message with TTL zero, stops forwarding the message. The pseudo-code for the Query Routing algorithm is illustrated in Figure 2.

```
HandleQuery (QueryRequest Query) {
  Results = doLocalSearch(QueryBloomFilter, LocalRepository);
  if ( Results != null )
      return Results;
  else {      // need to forward the query to other peers
      RankedList = getRank(ImmediatePeerList,
                          QueryBloomFilter, BloomFilterPlus);
      // Will send to a fraction of the immediate peers only
      ToSendList = getList(RankedList, PRate);
      sendQuery(ToSendList, Query);
    }
}

getRank (BloomFilter QueryBloomFilter,
                        BloomFilterList BloomFilterPlus){
   init RankedList to 0;
   do {
      Rank =0;
      for ( k=0 until BloomFilterWidth) {
         Bit = (QueryBloomFilter[k]) XOR (BloomFilterPlus.filter[k]);
         Bit = NEG(Bit);
         Rank = Rank + Bit;
      }
      BloomFilterPlus.filter = BloomFilterPlus.next;
      add Rank to RankedList;
   } // untill (have covered all bloomfilters in BloomFilterPlus)
   return RankedList;
}
```

Fig. 2. Pseudo code for Query Routing using Kundali

VI. UPDATE PROPAGATION

In our network, we consider that nodes decide individually what files to store at their local repositories. This regulates the rate at which updates are propagated to the peers. Typically, updates are generated when files are inserted, updated or deleted from the repository or when new peers join the network. Individual file updates are typically grouped into *group_updates* to further reduce the bandwidth consumption. The focal point of the update propagation algorithm is to try to propagate the local updates in such a way that the total amount of traffic in the network is kept to a minimal.

Per receipt of an Update message, the node checks whether the message corresponds to one of its immediate peers, in order to update its Remote Bloom Filter entry in the Kundali data structure. Otherwise, it creates a new entry for the new peer. Then, it computes the synopsis of its local Kundali and propagates the synopsis to its immediate peers. The algorithm ensures that peers with the same interests, which have a direct connection with that node, will get notified for the update with minimal latency. To avoid loops in update propagation, our algorithm detects duplicates, and stops disseminating the updates further.

Lets consider what happens when a node joins the network. As shown in Figure 1, node *a* connects to the network via peer *b*. Initially node *a* will have it Kundali initialized to just its local Bloom Filter. Node *a* will send its LBF(a) to node *b*, which will add this to its Remote Bloom Filter List. Node *b* then calculates a synopsis of all the filters in its Kundali, but excluding each time the filter for that particular direction in which the synopsis will be sent. Hence the synopsis Bloom Filter to be sent to peer *d* will not contain Bloom Filter that *b* had for direction of *d.* This is done in all directions *including* the direction from which the original Bloom Filter update came from. Synopsis is simply calculated by doing an *OR* of the bits from various Bloom Filter bit arrays. This is shown in Figure 1.

VII. IMPLEMENTATION

We used two scenarios to observe the working of our algorithm in the peer-to-peer network. In the first scenario, we evaluated the efficiency of our Kundali data structure by determining how accurately and quickly a peer can find the files in the network. In the second, we measured the reduction in the number of messages.

A. Network Setup

We conducted a real experiment using a network of 25 peers over 100 Mbit/s Ethernet and using the TCP/IP protocol in our local HP network. We used the Gnutella protocol [7] for establishing the connections among the peers and searching for files in the network. To construct the Kundali data structure, we implemented new update messages that propagate the RemoteBloom filters to the peers. The peers were implemented in the Java version 1.1.4 language.

The peers were organized in a bi-directional graph topology. The TTL parameter was set to 7. Each peer was connected to either 2 or 3 peers. This made it possible for us to trace the path of each query and check if the Kundali was directing the queries to the right nodes towards the destination. Moreover this also enabled us to the check the routing accuracy of Kundali.

B. Kundali Prototype

To minimize the probability of false positives in the network, our first objective was to determine optimal values for the m, k, and n parameters.

To keep the computational complexity low we used 3 hash functions. On an average each peer was connected to 3 more peers; and therefore the number of Remote Bloom Filters per peer was 3. A minimal false positive error is achieved by using $k = \frac{m}{n} \ln 2$ .Hence, by choosing a size of 128 bits per Bloom filter, we were able to represent about 800 unique files in our network distributed among the peers. These files could be further replicated at multiple peers. These numbers give us very few chances of getting false positives.

C. Performance metrics

We evaluated the accuracy of our Kundali data structure using the following metrics:

- *Propagation Rate (PRate):* defined as the percentage of peers to which to forward the request (Query) per hop. A PRate of 100% denotes that the Query message is propagated to all the immediate peers (the typical Gnutella search mechanism), while a PRate of 30% denotes that the Query is forwarded only to 30% of the immediate peers.
- *Popularity*: this represents the percentage of peers having a particular file and is computed as:

$$Popularity = \frac{numPeersHaveFile}{totalPeers} * 100$$

- *Redundancy*: defined as the percentage of excessive results returned (objects found) per search request. We assume that some redundancy is required, but it should be minimal. This distinguishes are system from others systems. Most of the existing system focuses on finding multiple responses for a request. But our goal is to obtain only few copies of the object. This goal is directed by the fact that we know the exact reference for that object. Hence the extra copies of the object returned are considered as '*redundant'* in our system. If the object is found, the request is not forwarded further in the network. Redundancy is computed as:

$$redundancy = \frac{numFilesFound}{totalNumFilesNetwork} * 100$$

```
gotBloomFilter (BloomFilter ReceivedBloomFilter,
                    HostAddress remotehost){
  if (remotehost in ImmediatePeerList)
       update <ReceivedBloomFilter, remotehost> in BloomFilterPlus;
  else
       new <ReceivedBloomFilter, remotehost> in BloomFilterPlus;

  BloomFilterToSend = calculateSynopsis(BloomFilterPlus,
                                        LocalBloomFilter, remotehost);
  send BloomFilterToSend to all peers in ImmediatePeerList;
}

calculateSynopsis(BloomFilterPlus, LocalBloomFilter, remotehost){
  do {
     HostAddress = ImmediatePeerList.address;
     bf = get(bloomfilter for HostAddress from BloomFilterPlus);
     if ((HostAddress != remotehost) && (bf != null)) {
         for (j=0 until BloomFilterWidth)
             synopsis[j] = synopsis[j] OR bf[j];
     }
   } untill (all peers in ImmediatePeerList covered)
   //OR with local bloomfilter
   for (j=0 till BloomFilterWidth){
       synopsis[j] = synopsis[j] OR LocalBloomFilter[j];
   }
   return synopsis;
}
```

Fig. 3. Pseudo code for Update Propagation using Kundali

We studied the effect of our algorithms for the following three object allocation scenarios (related to how close the files are to the requesting peers):

- *Clustered*: The objects were located close to each other in the network.
- *Random::Close to Sender*: The objects were scattered randomly in the network, but were only few hops away from the origin of the query.
- *Random::Far from Sender*: The objects were scattered randomly in the network, with most of the objects far away from the origin of the query.

The above scenarios allowed us to demonstrate the efficiency of our Kundali mechanism for different distributions of the files in the network.

D. Analysis

In the first set of experiments we evaluated the accuracy of our Kundali data structure by measuring the percentage of

data objects we find in the network in each of the three different allocation strategies described above.

Figure 4 shows the redundancy of the objects as a function of their popularity in the *Clustered* allocation scenario. Each graph in the figure represents a different propagation rate at the peers. For example, 30% indicates that the node sends the Query to only about one third of the peers, which in our case was 1 out of an average of 3 peers. Our first observation is that our mechanism was able to discover the objects at the peers efficiently in most of the cases. When the PRate is high (100%, 70%) the objects are always retrieved. When the PRate was 30% and at low popularity ratio, there was a case when we did not get any responses at all. That was because of false positives. The Kundali chose to follow the wrong path and could not retrieve the object. Over time, and as more peers cache or replicate the objects, the objects get closer to the requested peers, and therefore, the accuracy of our Kundali mechanism increases.

The *Random::Close to Sender* allocation strategy depicts similar results, as shown in Figure 5. In this experiment we were able to find the objects at all propagation rates even though the error probability was non-zero. With a PRate of 30%, the Redundancy was 30% i.e we were able to find on an average about 30% of the objects in the network. For the 70% and the 100% case this number becomes 57%.

The *Random::Far from Sender* allocation strategy depicts the most interesting results (Figure 6). This is the case where the objects are scattered randomly in the network, far from the requesting node. With propagation rates of 100% and 70% we were able to retrieve more than 60% of the objects at all times, similar to the previous cases. Notice, though, that the probability of getting false positives increased when the propagation rate was 30%. This stayed the same, even when the popularity of the objects was increased. This indicates that the distribution of the objects along with the topology of the network greatly affects the routing decisions made by our algorithm. However, the gradual movement of the objects of interest closer to the requesting nodes improves the error probability and maximizes the probability that the objects are found in the network.

When using Kundali for routing the query (i.e. cases when PRate was 70% and 30%) as seen from Figure 5 and 6, the *Redundancy* value decreases. Note that the query contains the exact file identifier for the file that we need to search. Hence even a single successful query hit message will suffice. For the case where the PRate is 30%, the Availability is on an average about 20%. This in most cases should be enough.

As one can notice that some of the points in the graphs appear to follow a unique curvature compared to rest of the points. This is due to the way the objects were placed in the network. Also since we used only 25 peers to test our prototype, the graph curvature is not smooth. Simulation for testing peer-to-peer network is an option that we did think about, but reliability of such simulations can sometime vary considerably from the actual implementation driven from real applications.

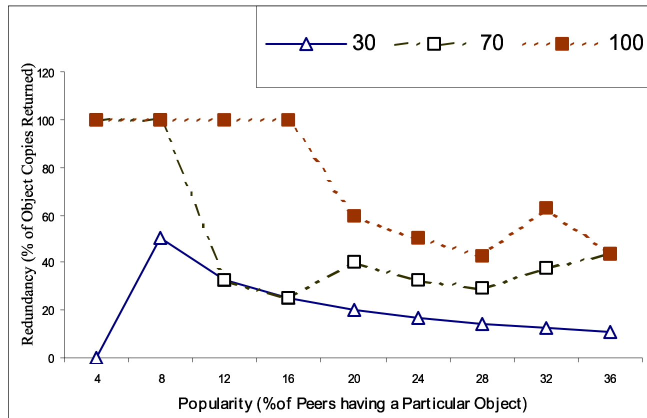

Fig. 4. Redundancy vs. Popularity for Different PRate for the case when Replicas are Clustered.

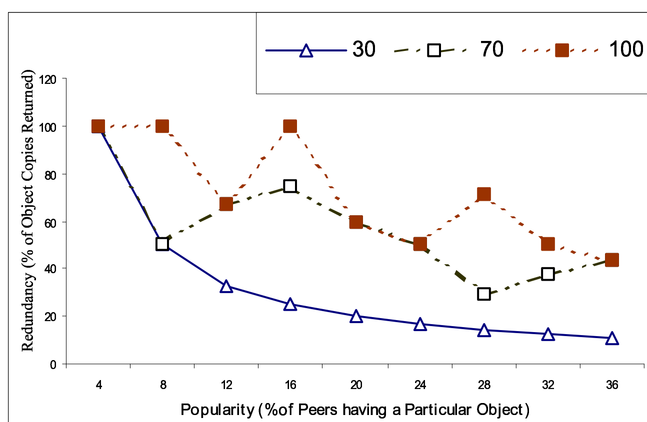

Fig. 5. Redundancy vs. Popularity for Different Prate for the case when Replicas Scattered Randomly but is close to the Query Origin.

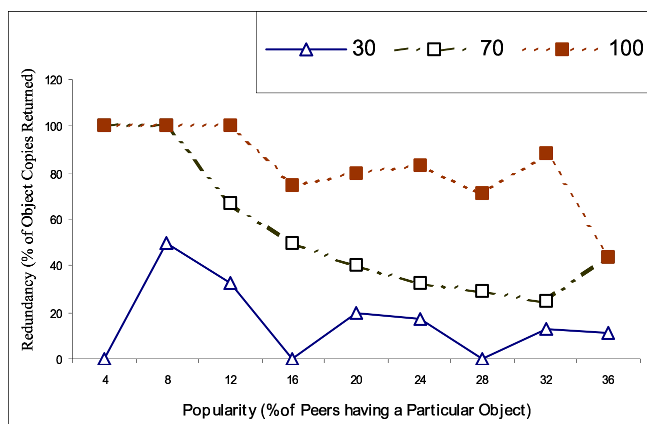

Fig. 6. Redundancy vs. Popularity for Different Prate for the case when Replicas Scattered Randomly but are far from the Query Origin

In the second set of experiments we measured the bandwidth reduction in the network. Figure 10 shows the percentage of reduction in the total number of messages in the network using a PRate of 30%, compared to an exhaustive search (when PRate = 100%). The figure shows that the total bandwidth reduction is at least 60% in all object allocation strategies. For the case when the objects are placed randomly in the network (either close or far from the origin of the query), the gain in bandwidth reduction is even more – around 80%. Hence our approach greatly reduces the network bandwidth without degradation in the performance.

VIII. RELATED WORK

The current distributed searches in pure peer-to-peer networks [7] use a brute force algorithm and broadcast the search requests to all the peers. This mechanism is not efficient as the messages have to travel a large number of hops from one peer to another to find the results and therefore has the potential of flooding the network. To improve the effectiveness of the blind searches, Lv et al [9] propose random walks that reduce the network traffic. Techniques for efficient searching in P2P networks have been proposed recently in the literature [14,17]. In these, a node propagates the query to some of its peers based on aggregated statistics such as which peer was the last to answer a query. Our routing mechanism is driven by the actual data stored at the peers and the updates generated by the peers to probabilistically move the requests closer to the results.

Recent work has studied the single problem of location and routing in building global-scale persistent storage utility centers [8, 12, 13]. Similar to our approach, Rhea and Kubiatowicz [11] propose a probabilistic location protocol based on attenuated Bloom filters, which improves the latency of locating files. The difference from our mechanism is that they locate the files to specific nodes based on some keys and use these keys to route the requests to the nodes. Moreover, Aspnes et al [18] have shown that there is no need for global coordination in the network. Our approach has the advantage that does not impose any structure; we assume that the network is self-organizing driven only by local incentive-based decisions that are made individually by each node.

The Bloom Filter mechanism has also been studied by Fan et. al. in the context of caching protocols for relieving hot spots on the WWW [4]. They demonstrate that that Bloom filter representations are economical and reduce the bandwidth consumption in the network. Our work builds upon these results and demonstrates that the Bloom filter approach can efficiently being used for propagation and routing in a highly dynamic network subject to frequent node arrivals and departures.

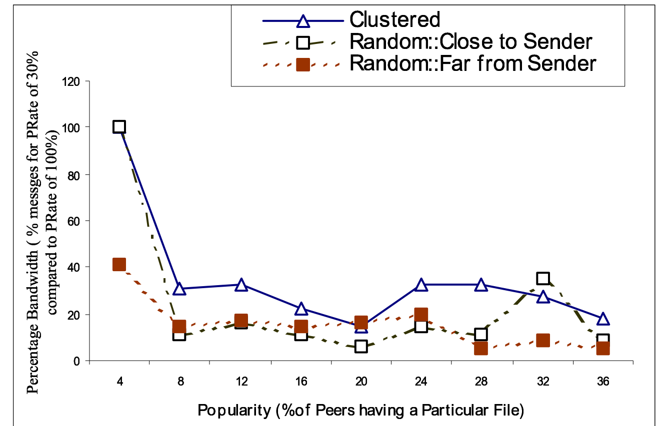

Fig. 7. % Percentage Bandwidth (request + reply) when using Kundali

IX. CONCLUSION AND FUTURE WORK

In this paper we propose routing and update propagation mechanisms in self-organizing peer-to-peer networks. This has the important advantages of adaptation and load balancing, high availability and good performance. Our mechanism builds bloom filter based synopsis that eliminate the need to monitor the messages propagated in the network or to send entire document lists among the peers. The mechanism has been used and tested in our internal HP peer-to-peer network. As for future work, we would like to continue evaluating a network of this sort with more diverse topologies and applications, and build upon our results to employ caching techniques that minimize the network traffic and improve the end-to-end latency.

REFERENCES

[1] B. Bloom, “Space/time trade-offs in hash coding with allowable errors.,” in *Communications of the ACM*, July 1970, vol. 13(7), pp. 422–426.

[2] Mitzenmacher, M., Compressed Bloom Filters. in *Twentieth ACM Symposium on Principles of Distributed Computing (PODC 2001)*, (Newport, Rhode Island, 2001).

[3] Mohan A., “Distributed Searching in Mobile Peer-to-Peer Systems”, in “AD-HOC Networks and Wireless (ADHOC-NOW), Sep. 2002, Toronto, Canada.

[4] Fan, L., Cao, P., Almeida, J. and Broder, A., Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. in *Proceedings of ACM SIGCOMM'98*, (Vancouver, Canada, 1998).

[5] K. Eshghi, , Intrinsic References in Distributed Systems, *HP Labs Technical Report, HPL-2002-32* (2002).

[6] Freenet Home Page, http://freenet.sourceforge.com

[7] Gnutella Home Page, http://ww.gnutella.com

[8] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, OceanStore: An Architecture for Global-Scale Persistent Storage, *Proceedings of ASPLOS*, Cambridge, MA (2000).

[9] Q. Lv, P. Cao, E. Cohen, K. Li and C. Shenker, Search and Replication in Unstructured Peer-to-Peer Networks, Proceedings of the 16th Innternational Conference on Supercomputing, New York (June 2002).

[10] M. K. Ramanathan, V. Kalogeraki and J. Pruyne , Finding Good Peers in Peer-to-Peer Networks,", *International Parallel and Distributed Computing Symposium*, Fort Lauderdale, Florida (April 2002)

[11] S. C. Rhea and J. Kubiatowicz, Probabilistic Location and Routing, Proceedings of the INFOCOM 2002, New York (2002)

[12] A. Rowstron and P. Drusche, Storage Management and Caching in PAST, a Large-scale Persistent Peer-To-Peer Storage Utility, *Proceedings of the 18th SOSP*, Toronto, Canada (2001)

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, Proceedings of the SIGCOMM'01, San Diego, CA (August 2001)

[14] B. Yang, H. Garcia-Molina, Efficient Search in Peer-to-Peer Networks, *Proceedings of ICDCS 2002*, Vienna, Austria (July 2002).

[15] Napster home page, http://www.napster.com

[16] Morpeus home page, http://www.musiccity.com

[17] B. Yang, H. Garcia-Molina, Comparing hybrid peer-to-peer systems, *Proceedings of Very Large Databases*, Rome, Italy, September 2001.

[18] J. Aspnes, Z. Diamadi, G. Shah, Fault-tolerance Routing in Peer-to-Peer Systems, *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, CA (July 2002).