Evaluating GUESS and Non-Forwarding Peer-to-Peer Search

Beverly Yang Patrick Vinograd Hector Garcia-Molina

Abstract— **Current search techniques over *unstructured peer-to-peer* networks rely on intelligent forwarding-based techniques to propagate queries to other peers in the network. Forwarding techniques are attractive because they typically require little state and offer robustness to peer failures; however they have inherent performance drawbacks due to the overhead of forwarding and lack of central control. In this paper, we study GUESS, a *non-forwarding* search mechanism, as a viable alternative to currently popular forwarding-based mechanisms. We show how non-forwarding mechanisms can be over an order of magnitude more efficient than forwarding mechanisms; however, they must be deployed with care, as a naive implementation can reduce in highly suboptimal performance, and make them susceptible to hotspots and misbehaving peers.**

I. Introduction

Peer-to-peer systems have recently become a popular medium through which to share huge amounts of data. Because P2P systems distribute the main costs of sharing data – disk space for storing files and bandwidth for transferring them – across the peers in the network, they have been able to scale without the need for powerful, expensive servers. For example, as of May 2003 the KaZaA [1] file-sharing system reported over 4.5 million users sharing a total of 7 petabytes of data.

The key to the usability of a data-sharing peer-to-peer system is the ability to search for and retrieve data efficiently. The best way to search in a given system depends on the needs of the application. For example, DHT-based search techniques (e.g., [2], [3], [4]) are well-suited for file systems or archival systems focused on availability, because they guarantee location of content if it exists, within a bounded number of hops. To achieve these properties, these techniques tightly control both the placement of data among peers and the topology of the network, and currently only support search by identifier. In contrast, other mechanisms, such as Gnutella [5], are designed for more flexible applications with richer queries, and meant for a wide range of users from autonomous organizations. These search techniques must therefore operate under a different set of constraints than techniques developed for persistent storage utilities, such as providing greater respect to the autonomy of individual peers.

We are interested in studying the search problem for these "flexible" applications because they reflect the characteristics of the most widely used systems in practice. Most of the research in this area has focused on *forwarding-based* techniques, where a query message is forwarded between peers in the overlay until some stopping criterion is met. Different refinements of forwarding-based techniques have been studied, such as arranging good topologies for the overlay [6], [7], intelligent forwarding of messages within the overlay [8], [9], [10], the use of lightweight indices [11], data replication [12], and security [13].

Despite the success of the above research in showing how forwarding-based techniques can be effective, some of the results also raise the question of whether message forwarding is truly necessary. For example, message-forwarding makes it difficult to control how many peers receive the query message, and which peers receive it, since there is no centralized point of control to monitor and guide the messages. However, references [9], [8] show that incremental forwarding of query messages and intelligent peer selection greatly improves search performance without affecting quality of results.

In this paper, we wish to to investigate a new type of search architecture, in which messages are *not forwarded*, and peers have complete control over who receives its queries and when. We are currently studying this non-forwarding architecture in the context of the GUESS [14] protocol, an under-construction specification that is meant to become the successor of the widely-used but inefficient Gnutella protocol. Under the GUESS protocol, peers directly probe each other with their own query messages, rather than relying on other peers to forward the message.

However, the GUESS protocol is being designed without a good understanding of the issues and necessary strategies to make it work. For example, when processing a query, in what order should peers be probed? The solution to this "peer selection" problem must balance efficiency of the query with load-balancing among the peers. Also, if messages are not forwarded, then a peer must know of many other peers (rather than just a handful of neighbors) in order to successfully find answers to its queries. How should this large state be built up and maintained? Practical problems not directly related to search performance must also be addressed; for example, since peers no longer rely on other peers to forward their queries, it is much easier for peers to abuse the system for personal gain. How can we detect and prevent selfish behavior? We are currently investigating solutions to these and other issues to make GUESS a viable alternative to other proven P2P search protocols.

We note that the non-forwarding concept has also been proposed for one-hop lookup queries in DHTs [15]. Like [15], the purpose of GUESS is to reduce the overhead of message forwarding; however, because GUESS allows "flexible" search over loosely structured networks, the protocol and its underlying issues (e.g., how to maintain state, how to select peers to query, etc.) are very different.

In this paper, our goals are to promote the concept of a non-forwarding search mechanism for flexible search, understand what the tradeoffs are compared to existing forwarding-based techniques, and investigate how the GUESS non-forwarding protocol can be optimized. In particular, our contributions are as follows:

- We present an overview of the GUESS protocol (Section II), based on the specification written by the Gnutella Development Forum [14].
- We identify the importance of *policies* in the performance of a non-forwarding protocol, and introduce several policies that are feasible to implement in a real system, and that might accomplish reasonable goals such as fairness, freshness, efficiency of search, etc.
- Using simulations, we demonstrate how GUESS, if implemented in a straightforward way, can have serious performance problems. For instance, we show how careful choice of policy can improve performance dramatically (Section VI-B), but that a naive choice can result in a mechanism that is unfair (Section VI-C), and not robust to cheating peers (Section VI-D).

In this paper our focus is on the performance of the GUESS protocol in the face of cooperating peers, and misbehaving but non-colluding peers. Reference [16] (also submitted to ICDCS) focuses on a broader range of security issues in the GUESS protocol, in the face of malicious peer collectives.

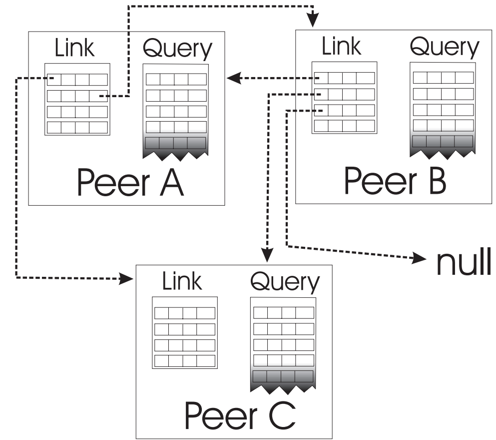

Fig. 1. Illustration of a small GUESS network. Note that peer A points to peer C, but C does not point back to A; peer B has one entry pointing to a non-existing peer. Although neighbor pointers do not actually represent open, active connections between peers, they still form a "conceptual" overlay network, as illustrated in Figure 2

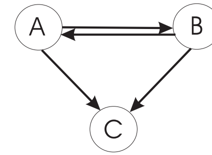

Fig. 2. Conceptual overlay representation of the GUESS network in Figure 1

II. GUESS Protocol

In this section we describe the GUESS protocol for querying and state maintenance. For more details, please refer to the original specification [14]. Some of the information in this section is not part of the original protocol (e.g., the format of a cache entry), but are implementation details added for clarity.

A. Basic Architecture

Peers running the GUESS protocol will maintain two *caches*, or lists of pointers (IP addresses) to other peers: a *link cache*, and a *query cache*. The link cache is analogous to a peer's neighbor list in Gnutella; all peers appearing the link cache of a peer P can be considered P's neighbors. Rather than keeping an open TCP connection with each neighbor, however, P will communicate with neighbors via UDP. Hence, the "neighbor" relationship is one way: if Q appears in P's link cache, P might not appear in Q's link cache. Furthermore, because the UDP protocol does not maintain an active connection between two hosts, it is possible for a peer's neighbor to die without the peer's knowledge. We discuss the issue of maintaining neighbor pointers in Section II-B. Please refer to Figure 1 for an illustration of a GUESS network.

The *query cache* is simply a "scratch space" to temporarily hold large number of pointers to other peers in order to improve query performance. We discuss the use of the query cache further in Section II-C.

An *entry* in the link or query cache, essentially a "pointer" to some peer Q, has the following format:

$$\{\text{IP address of } Q, TS, \text{NumFiles}, \text{NumRes}\} \quad (1)$$

The TS field holds the timestamp of the last interaction with peer Q. When P interacts with Q, regardless of which party initiated the interaction, P will update the TS field in its cache entry for Q, if such an entry exists. The NumFiles field holds the number of files being shared by Q. This field is set by Q when it first "introduces" itself to the network, and is passed on as cache entries are shared (introduction and cache entry sharing are discussed in Section II-B). Similarly, the NumRes field holds the number of results last returned by Q. Each time peer P sends a query to peer Q, it resets the value of NumRes according to Q's response to that query.

B. Maintaining State

Because peers in P2P systems typically have short lifetimes [17], peers must actively make sure the entries in their link cache are fresh. Otherwise, over time a peer's link cache will accumulate the addresses of many dead peers, which can result in poor query performance for that peer, and fragmentation of the "conceptual overlay."

A peer maintains its link cache by periodically selecting an entry and sending a *Ping* message to the neighbor. If the neighbor does not respond, the peer will evict this entry from its cache. If the neighbor does respond, then the peer will update the TS field of the cache entry. Note that given a fixed effort from the peer to maintain its link cache, the rate at which a given cache entry is pinged is inversely proportional to the size of the cache. Therefore it is important that the cache not be too large; otherwise, it cannot be properly maintained.

When a peer receives a Ping message, it will respond with a *Pong* message. A Pong message contains a small number

of IP addresses selected from the peer's own link cache. The purpose of Pong messages is to allow peers to *share* cache entries with each other. Sharing entries helps peers discover new live, productive peers to place in their cache. When a peer receives a Pong message, it will decide whether to add some or all of the entries to its link cache, depending on the cache replacement policy in use. If the peer does decide to place an entry from the Pong message into its own cache, it does not update any of the fields (i.e., TS, NumFiles, or NumRes).

When a new peer first joins the network, it does not appear in any other peers' link caches. An *introduction protocol* is needed to bring the IP address of new peers into the link caches of existing peers. For our purposes, we assume that when a peer P initiates an interaction with peer Q by send either a Ping or Query message, then Q will add P to its cache with some probability p. Note that it is important that $p < 1$; otherwise it would be very easy for malicious peers to infiltrate the caches of many good peers (see Section VI-D for a discussion of "cache poisoning"). A new peer will therefore be added to existing peers' caches with some probability as soon as it initiates a query or ping. Once the new peer appears in other peers' caches, it can be circulated even further via Pong messages.

C. Query Propagation

The essential characteristic of GUESS is that Query messages are not propagated via flooding-based broadcast. Instead, a GUESS peer simply iterates through the entries in its link cache, and performing a unicast query, or *probe*, to the target peer. A peer should probe only as many other peers as necessary to obtain a sufficient number of results. Also, the GUESS protocol specifies that query happens in a strictly serial manner; after sending a probe, a peer must either receive the reply or wait for *timeout period*, before it may probe the next neighbor.[1]

In some cases, a querying peer must be able to probe a large number of peers to receive satisfactory results. However, the number of addresses that a peer can actively keep track of is limited by the size of the link cache. As we discussed in the previous section, the link cache must be relatively small if we are to maintain it properly.

To counter this problem, when a peer is probed, it returns a Pong message in addition to any results to the query it may find. Then, the querying peer places the entries from this Pong message in its *query cache*, which is a temporary cache of (theoretically) unbounded size. Entries in the query cache have the same format as link cache entries. The querying peer may probe peers from either its link cache or its query cache. In this manner, a peer is able to probe a much larger number of peers than it can maintain in its link cache. Entries in the query cache are not maintained after the query is completed, otherwise, maintenance overhead would be too high. However, qualifying entries may be inserted into the link cache, depending on the cache replacement policy in use.

[1]Parallel probes are also possible, although the current GUESS specification makes no such provisions.

III. GUESS VS. GNUTELLA

The Gnutella network, which users a forwarding-based search mechanism, probably provides the nearest data point in terms of understanding the operation and intended use of GUESS. Here we touch on some of the high-level points of comparison between these protocols, as a generic comparison of forwarding versus non-forwarding techniques (as opposed to exact protocol details). A more detailed discussion of these issues can be found in our extended report [18].

Query Performance: In Gnutella, the number and identity of peers that a query reaches is largely fixed, determined by the flooding query mechanism and the location of a peer within the overlay network. In GUESS, on the other hand, a peer has control over the order in which it probes peers; furthermore, it can also decide how many peers to probe, matching the extent of the query to how hard the file is to find. These two decisions can have a great effect on query efficiency; we examine the effect of both of these decisions in Section VI-B. One drawback to such freedom is the possibility of *hotspots*, where many peers all try to probe the same productive peer, exceeding that peer's capacity. We examine this issue further in Section VI-C. The tradeoff of probing nodes in a serial manner is poor response time; possible solutions include introducing some parallelism, or adjusting the probe rate adaptively according to how many results are found.

State Maintenance: The state of a Gnutella peer consists of a small number of active network connections to the peer's neighbors. The overlay provides a highly consistent structure even though a given peer might only be aware of a small part of that structure. In GUESS, however, each peer must maintain pointers in a large link cache; while the size of such state is well within available memory limits, maintaining the cache over time can require a lot of network bandwidth; we examine the cost of maintaining state in Section VI-A. The GUESS network is also much less consistent; peers may not have mutual knowledge of one another since link cache pointers are one-way. Similarly, when a peer joins or leaves the network, there is no explicit notification to other peers. Instead, other peers must be made aware of the change by some introduction mechanism, or by wasting a probe trying to contact a dead peer, respectively.

Security: There are two types of misbehavior we consider with GUESS: *malicious* peers who try to make the system unusable, and *selfish* peers who try to "game" the system in their favor. Gnutella is fairly robust to selfish peers because once a peer has sent out its query, it is dependent on other peers to propagate it. GUESS peers, on the other hand, can easily probe many peers at a time, which improves response time while imposing a higher load than necessary on the system.

As for malicious behavior, Gnutella is vulnerable to Denial of Service (DoS) attacks, by virtue of the traffic amplification effect of the broadcast query mechanism [13]. Since GUESS does not magnify queries in this way, a GUESS peer can only cause as much network traffic as it itself is able to initiate. Fragmentation attacks, which attempt to fragment the overlay network, are a risk in both Gnutella [17] and GUESS.

In Gnutella, highly-connected peers are attacked; in GUESS, groups of malicious peers can propagate their identities aggressively into many link caches; if they then suddenly disappear, the conceptual overlay of the remaining nodes will become fragmented. The security concerns of GUESS are discussed further in Section VI-D.

IV. Policies

We observe that performance of the GUESS protocol depends heavily on the *policies* that determine how entries in the pong cache are used and maintained. For example, by constructing a Pong message using entries with the latest timestamps, and evicting entries with the oldest timestamps, peers might be able to maximize the number of live entries in their cache. As another example, by first probing peers who have a history of providing useful information, peers can drastically reduce the total number of probes needed to answer a query. Hence, any deployment of the GUESS protocol must first carefully consider which policies to implement.

There are five *types* of policies which we must consider:

- `QueryProbe` – the order in which peers in the link and query caches are probed for queries
- `QueryPong` – the preference given to entries when constructing a Pong message in response to a Query
- `PingProbe` – the order in which peers in the link cache are pinged
- `PingPong` – the preference given to entries when constructing a Pong message in response to a Ping
- `CacheReplacement` – the order in which peers are evicted from the link cache

We consider QueryProbe and PingProbe (and QueryPong and PingPong) separately because a peer might have different goals during a query and during a ping. For example, during a query, a peer may prefer to probe other peers who are likely to have a file. For a ping, however, a peer may prefer to probe other peers who have many link cache entries, or are known to be alive.

For each of the policy types discussed above, many policies could be implemented. For the purposes of our experimentation, we came up with a number of policies that we felt would be feasible to implement in a real system, and that might accomplish reasonable goals such as fairness, freshness, efficiency of search, etc. The policies that we implemented are listed below along with a brief discussion of the rationale for each policy.

Note that for Cache Replacement, the policy name indicates what peers get evicted from the cache. Therefore, to get the same intended effect as, say, a Probe policy, we must reverse the criterion used. For example, to effect a Most Files Shared goal, we use a Cache Replacement policy of Least Files Shared, since by evicting those peers with a small number of files we retain the ones with more files. Similarly, Most Results becomes Least Results, and Least/Most Recently used become Most/Least Recently Used.

Random (Ran) – selects entries at random. This policy is used as a baseline for comparison, and is likely to be very fair in terms of load distribution.

Name	Default Value
NetworkSize	1000 peers
NumDesiredResults	1
LifespanMultiplier	1
Query Rate	$9.26 \cdot 10^{-3}$ queries/user/sec
MaxProbesPerSecond	100 probes/sec
PercentBadPeers	0%
BadPongBehavior	Dead
QueryProbe	Random
QueryPong	Random
PingProbe	Random
PingPong	Random
CacheReplacement	Random
PingInterval	30 sec
CacheSize	100 entries
ResetNumResults	No
DoBackoff	No
PongSize	5 entries
IntroProb	.1

TABLE I
System and protocol parameters, with default values

Most Recently Used (MRU) – prioritizes pong cache entries with the most recent timestamps. These entries are most likely to be alive since they have been in contact recently; therefore MRU should waste the least amount of work in probing dead peers.

Least Recently Used (LRU) – opposite of MRU, prioritizing cache entries that have old timestamps. The rationale behind LRU is *fairness*; rather than continually querying the same set of peers, load is spread across peers that have not been contacted recently. Of course, peers with a very old timestamp is more likely to be dead, resulting in in a wasted probes.

Most Files Shared (MFS) – prioritizes entries based on the number of files they share. The impetus for such a policy is obvious; peers with many files are more likely to be able to have files related to the query. A potential downside to this policy is that the measure used (files shared) is global, and so the peers with many files shared are likely to be queried by a large number of peers, making them shoulder an unfair amount of work in the network.

Most Results (MR) – similar in nature to Most Files Shared, prioritizes entries based on the number of good results that the corresponding peers have returned in the past. Typically, peers that have been fruitful in the past may be more likely to be good in the future. MR is less susceptible to lying than MR, although often less good at identifying useful peers.

One potential advantage of MR over Most Files Shared is that MR includes some notion of *personal* usefulness. A peer with many files may not have the files that I want; however, if the queries that I generate are related, then perhaps a peer that has worked well for me will continue to work well, regardless of its total number of files. Similarly, MR might be better than MFS at identifying peers who are actually capable of servicing queries, which is important when capacity limits come into play.

V. Experimental Setup

Parameters. We will be comparing different *configurations* of the GUESS protocol, where a configuration is defined by

a set of system and protocol parameters, shown in Table I. System parameters describe the nature of the system on which the GUESS protocol is used (e.g., the query behavior of the users). Protocol parameters then describe how the GUESS protocol is configured (e.g., the policy used to order query probes). As we will see in Section VI, different protocol parameters result in better performance in different system scenarios. Parameters will be described in further detail as they are used later. Unless otherwise specified, our simulations use the default values shown in these tables.

Note that NetworkSize=1000 is a modest number of peers, given the scale of the types of system we expect to use GUESS. In our extended report [18], we investigate how our results scale with network size; results shown here are representative, unless otherwise noted.

Metrics. The main metric of query efficiency is the average number of *probes per query* needed; minimizing probe traffic is one of the primary goals of the GUESS protocol. Of course, such a goal is only reasonable if users receive results for their queries, hence another key metric is the proportion of queries that go *unsatisfied* (i.e., do not return `NumDesiredResults` answers). We also examine the proportion of probes that are *wasted*; that is, sent to peers that have already left the network. For each peer, we also measure the number of *received* probes; comparing the load across peers help us to gauge fairness as well as efficiency.

Simulation Details. A detailed description of each parameter and its role in our simulation is provided in [18]. Here, we highlight the most relevant parameters and concepts in our simulation.

The simulation begins with `NetworkSize` live peers in the network. As time progresses, peers will die. A large sample of peer lifetimes in the Gnutella network was measured in [17]. For our simulations, the lifetime of a peer is drawn randomly from this sample. In addition, we may tune these lifespans via the `LifespanMultplier` parameter. If `LifespanMultiplier` $= x$, then all values in the measured distribution of lifespans are multiplied by x.

When a peer dies, we assume that it never returns to the system. This assumption is conservative in that it is the worst-case scenario for cache maintenance; our maintenance policies must be shown to be effective even in this worst case. Also, when a peer dies, a new peer is "born." In this way, there are always `NetworkSize` live peers in the system. When a peer joins the network, it must populate its link cache. We use the *random friend* seeding policy as described in [16]. Under the random friend policy, we assume that the new peer knows of one other peer, or "friend," that is currently alive. The new peer initializes its link cache by copying the link cache of its friend.

As discussed earlier, the `QueryProbe` and `QueryPong` policies determine the order in which entries are probed and included in Pong messages, respectively, during a query. `PingProbe` and `PingPong` are the analogous policies for pings. When new entries are added to the cache (e.g., because they were received in a Pong message), `CacheReplacement` determines the order in which entries are evicted. Pong messages include `PongSize` entries. To maintain its cache, each peer sends one ping message every `PingInterval` seconds.

When a peer is probed, to determine whether it returns a result for the query, we use the query model developed in [19]. Though this query model was developed for hybrid file-sharing systems, it is still applicable to the file-sharing systems we are studying. The probability of returning a result depends partially on the number of files owned by that peer; number of files owned are assigned according to the distribution of files measured by [17] over Gnutella.

Peers set a maximum number of probes per second that it is able or willing to handle, according to `MaxProbesPerSecond`. Although different peers may have different capacities, we assume that in the interest of fairness, all peers set the same load limit. A peer is *overloaded* if the number of probes it must process per second is greater than `MaxProbesPerSecond`. When a peer becomes overloaded, it "refuses" a probe, meaning it notifies the querying peer that it is overloaded, and that the querying peer should "back off" from probing it. We will discuss backing off in further detail in Section VI-C.

VI. Results

In the following section, we present the results of our experiments over a wide range of system and protocol configurations. We organize the results into four main categories. First, we investigate the issue of maintaining the link cache (Section VI-A) in the face of frequent peer downtimes. We then study the behavior of policies in various system scenarios: in Section VI-B, we study basic performance in the default usage scenario. We then investigate the fairness of policies and their ability to perform in the presence of limited peer capacity (Section VI-C), as well as the robustness of policies to misbehaving peers (Section VI-D).

A. Maintaining the Link Cache

One of the main differences between GUESS and Gnutella is the way connections are maintained. In Gnutella, a few links are maintained very closely – as soon as a link is broken, the client tries to establish a new one. On the other hand, in GUESS, a large number of links are maintained loosely. Here, we wish to investigate the effect of cache sizes on query performance. In the following experiments, `PingInterval` is fixed while `CacheSize` is varied. If `CacheSize` is small, then each entry in the cache will be maintained fairly closely; otherwise, entries are maintained loosely.

Fragmentation. A fragmented overlay is bad because it degrades query performance. Furthermore, unless there is some form of centralized boot-strapping server (e.g., pong servers such as those run by LimeWire [20] for Gnutella), the network is unlikely to heal.

We begin by looking at an extreme case to illustrate a point. Figure 3 shows the largest connected component of the overlay network (calculated at the end of each simulation) as the size of the link cache varies. In this figure, `PingInterval` is set to 300 (one ping per 300 seconds), `LifeSpanMultiplier` is set to .2, and `NetworkSize` = 200. Although peers can

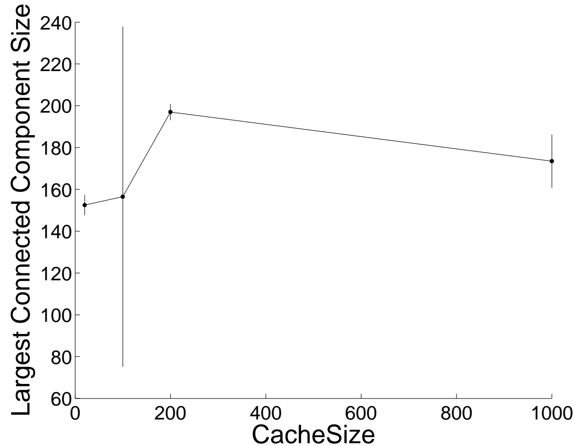

Fig. 3. Connectivity of overlay degrades with small or large cache sizes

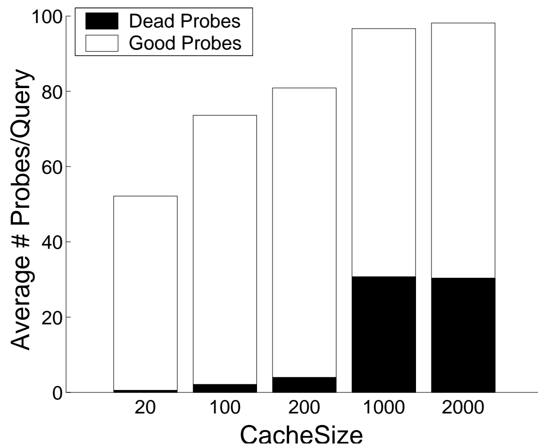

Fig. 4. Average probes per query increases as `CacheSize` increases

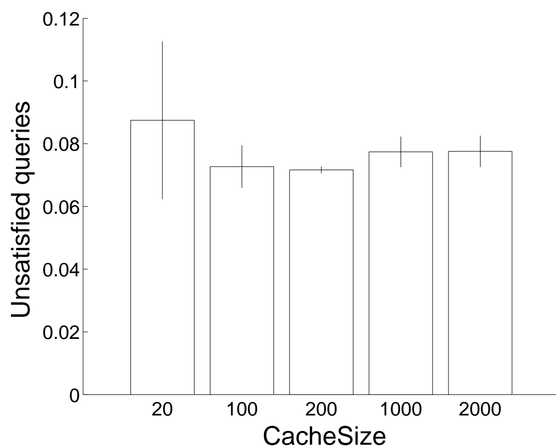

Fig. 5. Unsatisfied query rate reaches a minimum at a moderate `CacheSize` value

certainly support more than 1 ping per 300 seconds, this experiment is illustrative of the case where the network size is much larger, and cache size proportionally larger.

In Figure 3, we see that as expected, connectivity degrades when `CacheSize` is too small. When link cache is small, fewer peers need to die before the graph fragments. However, we also see that connectivity actually degrades when `CacheSize` is very large. Because cache entries are maintained so poorly, many links die before they can be replaced with live entries. Hence, the best choice for link cache size should avoid either extreme of being too large or too small – e.g., 200 entries in this example. Note that variance is very high at `CacheSize` = 100. Our experiments with networks of varying sizes show that at "moderate" values of `CacheSize` (e.g., roughly `CacheSize` = 100 to 200), the connectivity of the graph is either very good, or very bad. Further investigation is needed to determine why variance is high, and how cache size can be carefully selected so as to avoid high risk of poor connectivity.

Efficiency. Fresh links are important not only for network connectivity, but for efficiency and quality of results as well. In Figures 4 and 5, we see the average query cost and rate of unsatisfied queries, respectively, as `CacheSize` is varied. The parameters used for these figures are the same as in the previous figures, except that `PingInterval` = 30 and `NetworkSize` = 1000. At this value for `PingInterval`, the network is fully connected for all cache sizes. Cost of a query is measured in the number of probes, which is further broken down into wasted ("dead") probes to dead peers, and "good" probes to live peers. Notice in Figure 4 that the average number of probes is greater than the size of the link cache – the additional probes are enabled by Pong message entries that are inserted into the query cache.

Notice that smaller cache sizes result in less expensive queries. At the same time, however, with the exception of the extreme `CacheSize` = 20, smaller cache size also results in slightly lower unsatisfaction rates. The reason smaller cache size is more effectively is because the *ratio* of live to dead peers is higher when cache size is smaller. For example, at `CacheSize` = 100, the ratio of live to dead peers is almost 8:1, whereas at `CacheSize` = 1000, the ratio is 4:3. The more dead entries there are relative to live entries, the more wasted probes a peer will perform (as seen in Figure 4). Furthermore, since Pong messages will contain more dead IPs if peers' link caches contain more dead IPs, the total number of useful query cache entries will decrease as well. Hence, satisfaction rate degrades.

In summary, the size of the link cache must be carefully chosen to maximize query performance. The connectivity of the overlay, as well as the ratio of live to total link cache entries must both be considered. From our experiments, including but not limited to those represented in these figures, we find that a good link cache size falls at roughly `CacheSize` = `NetworkSize`/2. However, further experiments are necessary to see whether this rule of thumb scales to extremely large networks (e.g., millions). Due to the impracticality of simulating such a large network, as future work we wish to use analytical models to determine good link cache sizes.

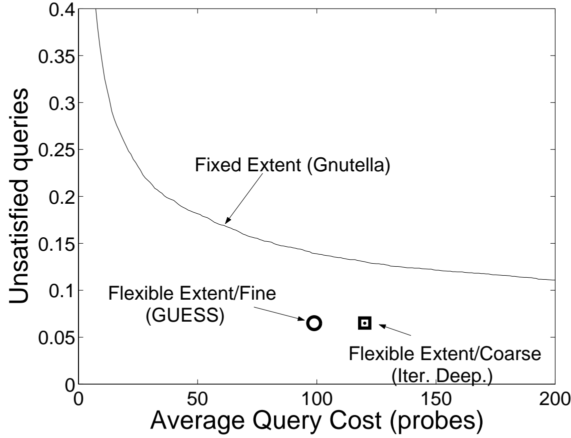

Fig. 6. For a given average query cost, unsatisfaction rate is lowest with a fine-grained flexible extent provided by GUESS

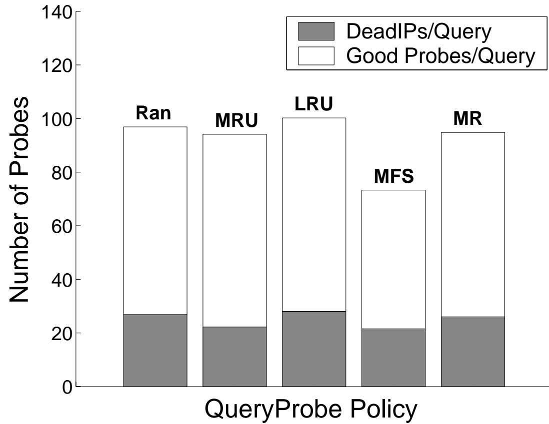

Fig. 7. Probes/Query for different QueryProbe policies

Ideally, we will find that the best value for `CacheSize` grows sublinearly with `NetworkSize`. In the future we would also like to investigate how link cache size can be chosen dynamically.

B. Basic Policies

Flexible Extent: First, we highlight the performance benefits of a completely flexible query extent. Figure 6 shows the tradeoff curve between the average cost of a query and the unsatisfaction rate for three different search mechanisms: a fixed-extent mechanism (representative of Gnutella), a coarse-grained flexible extent mechanism (representative of an iterative deepening [8] approach), and a fine-grained flexible extent mechanism (representative of GUESS). The iterative deepening mechanism sends, on each iteration, a query to all peers that are h hops away in a Gnutella-style overlay, starting from $h = 1$ and ending at $h = max$ (where max is a system-wide constant), and terminating at the earliest h that satisfies the query. Because the granularity of control it has over the extent is limited, we call it a "coarse-grained" flexible extent mechanism. In Figure 6 we show the performance of iterative deepening for $max = 8$ – the value of max necessary to achieve the best rate of unsatisfied queries – marked by a

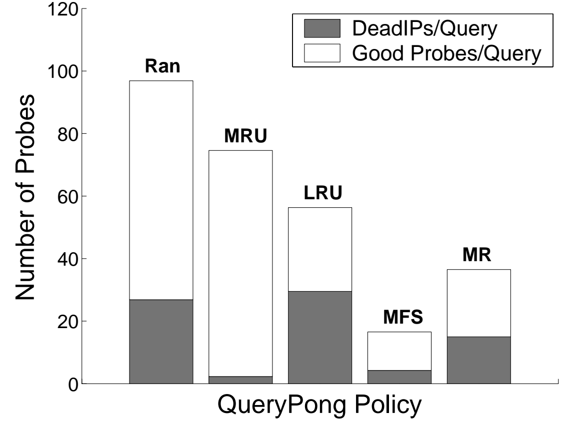

Fig. 8. Probes/Query for different QueryPong policies

square. The 'o' marks the point in the figure that represents GUESS, using the Random baseline policy for all policy types. Finally, for the fixed-extent mechanism, we evaluated the rate of unsatisfied queries for all fixed extent sizes from 1 to 1000, in order to view the tradeoff between efficiency and quality; hence, the fixed extent mechanism is represented by a curve in Figure 6. Please refer to [18] for a detailed description of how we modeled each of these three mechanisms.

From Figure 6, we can see the enormous performance gains that a flexible extent allows. For example, GUESS can achieve an unsatisfaction rate of almost 6% with an average query cost of 99 probes. In contrast, a fixed extent mechanism such as Gnutella would require 1000 probes for the same rate of unsatisfied queries – over an order of magnitude higher than GUESS. At a fixed extent size of 99, where average query cost is the same as for GUESS, the rate of unsatisfied queries would be over twice as high as with GUESS.

The reason fixed extent performs so poorly is because some queries are for popular items while others are for rare items, and the fixed extent can not adapt to these two extremes. For many queries that are for popular items, far mare peers receive the query than is necessary to satisfy the query. However, if the fixed extent is made small, then the queries for rare items can not be satisfied. Having a flexible extent allows one to probe just as many peers as necessary.

Because iterative deepening provides a flexible extent, like GUESS, it strikes a fairly good balance between cost and quality. However, its average query cost is still over 20% more expensive than average cost in GUESS. The reason for this higher cost is that the granularity of control over the number of peers who receive the query is coarse-grained; therefore, often times, many more peers receive the query than is necessary to satisfy it.

Bear in mind that the response time for a fine-grained flexible-extent mechanism such as GUESS can be a real problem. Because each probe is performed sequentially, for certain queries a user may have to wait minutes, in a naive implementation, before receiving a result, or finding that none exists. An easy partial solution is to allow for k parallel probes, thereby reducing response time by a factor of k while increasing cost by at most k probes. A more complete solution,

in which the degree of parallelism adapts to the current state of the query, is left as future work.

In summary, even the baseline Random policies for GUESS far outperform the baseline policy for Gnutella. In the next subsection, we will see how careful choice of policy can further improve the performance of GUESS by several factors over the Random policies.

Query Efficiency: Here we examine the many options available for the various policies, with the goal of seeing which choices for a given policy are the most effective in the standard usage scenario (i.e. no capacity limits, no malicious behavior). In particular we measure the number of probes used for queries and the percentage of queries that go unsatisfied. In terms of the number of probes used for queries, we distinguish between "useful" probes (those that get sent to live peers) and "wasted" probes which are sent to dead peers and therefore have no chance of returning useful information.

To simplify presentation of the results, in the rest of this section we fix the `PingProbe` and `PingPong` policies at Random so that we can focus on query behavior. Also, in each of the graphs presented, the unspecified parameters use the default values in Table I. In particular, aside from the policy being varied in the graph, the other policies are fixed as Random.

Looking at Figures 7, 8, and 9, our immediate observation is that choosing different policies can have a dramatic impact on performance. For example, in Figure 8 we see that changing the `QueryPong` policy can reduce cost (Probes/Query) by a factor of four. Likewise, in Figure 9 we see that changing the `CacheReplacement` policy can reduce cost by over a factor of five. The `QueryProbe` policy does not appear to make as significant a difference in performance compared to other policy types; changing the `QueryProbe` policy results in at most about a 25% change in cost. Certainly the `QueryProbe` policy should be chosen appropriately; however, as a first cut we recommend focusing attention on the other two policy types.

Another initial observation is that there are some policies with very serious drawbacks. The most obvious of these is the MRU policy for `CacheReplacement`; in Figure 9, we see that this policy causes a large number of probes to dead peers. This follows, since the policy evicts recently-contacted peers from its link cache, leaving the most stale entries. It appears that using MRU as a mechanism to enforce fairness does not result in effective search.

Looking at what does provide effective search, we see that the MFS `QueryPong` and LFS `CacheReplacement` policies are the most efficient (Figure 8 and Figure 9). In fact, used together, these policies result in query efficiency that is almost an order of magnitude better than if Random policies were used, thereby highlighting the importance of carefully selecting good policies. The MFS/LFS policies cause peers to circulate and maintain the identities of peers who share many files and are therefore more likely to return results to a query. The results-based policies (MR and LR) behave similarly, but are not quite as effective, because the number of results returned is not as accurate an indicator as number of files shared.

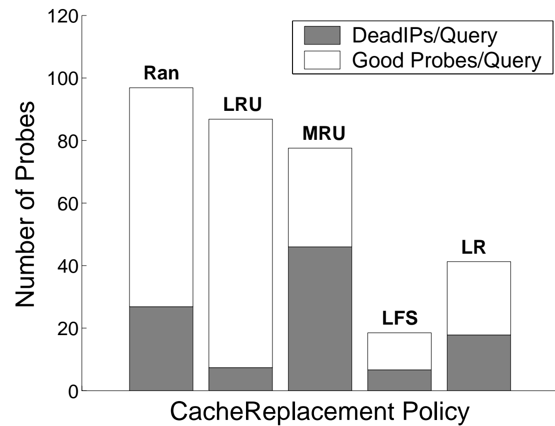

Fig. 9. Probes/Query for different CacheReplacement policies

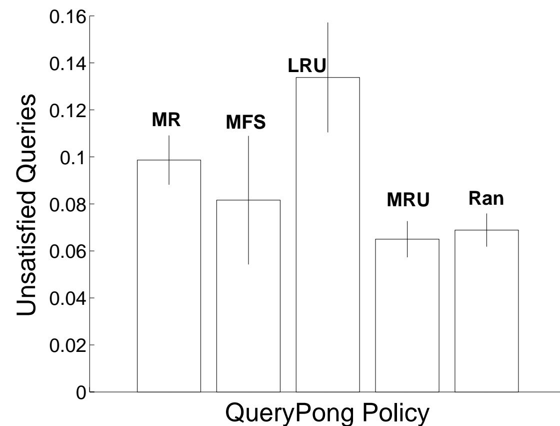

Fig. 10. Percentage of queries that are not satisfied, for different `QueryPong` policies

Unsatisfied Queries: Examining Figure 10, we observe that the proportion of unsatisfied queries is typically in the range of 6-14 percent. This figure may seem rather high given that one of the supposed advantages of GUESS is that it searches more extensively for rare files. This rate is partially an artifact of our simulation parameters; when simulating a network of only 1000 nodes, approximately 6% of the queries will go unsatisfied even if the entire network is probed, because some queries are for very rare or nonexistent items. In light of this effective lower bound on the unsatisfied query rate, we see that policies such as MFS and Random do quite well. An important aspect of satisfying queries is effectively circulating the identities of peers, since a peer will only stop searching when it runs out of other peers to probe. We discuss peer circulation in more detail in our technical report [18].

C. Individual Loads

Given that one of the problems encountered by the original Gnutella protocol was congestion, it makes sense to examine whether GUESS might be susceptible to such problems as well. While GUESS does not cause queries to be magnified, as in Gnutella, by a flooding mechanism, there may be other ways in which the limited capacities of peers come into play. We first investigate how different policies may lead to a high load

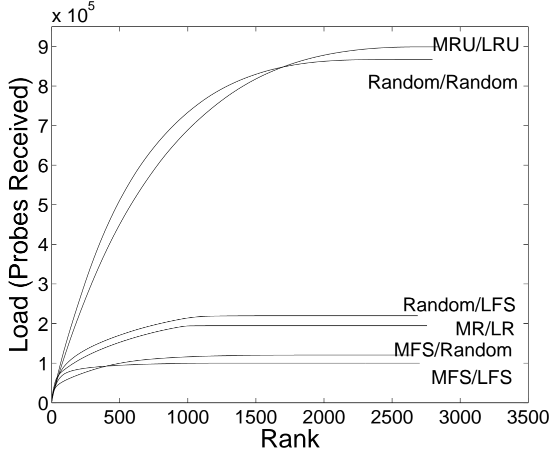

Fig. 11. Cumulative distribution of load (received probes) for different policies

for some peers, and then examine possible ways to remedy such a condition.

Fairness: Figure 11 shows the cumulative distribution of load (received probes) for several combinations of `QueryPong` and `CacheReplacement` policies. Peers are ranked by number of received probes, and the curve at a given point shows the total probes received by all the peers up to that rank. Thus, the point at which a curve levels off shows how many peers are handling the bulk of the load. We see that the policies based on results and files shared force just a few nodes to do most of the work; in particular the MFS/LFS combination levels off almost immediately. On the other hand, policies like Random/Random and MRU/LRU distribute the load much more evenly. No policy is going to be perfectly fair, mostly since the nodes have different lifespans.

To provide some further detail, the most heavily loaded peer in the MFS/LFS case receives 10.9% of the overall probes sent during the simulation; the most heavily loaded peer in the Random/Random case receives about 0.3% of the total probes. So, the Random/Random policy combination is more fair. On the other hand, it is less efficient. As seen by the height of each curve, Random/Random sends over 8 times as many probes than MFS/LFS. In the end, the goal of fairness may be trumped by other concerns, such as overall efficiency. However, this example does show that peers in the network may encounter very high loads, so we should determine how the system will react if the capacity of these peers is exceeded.

Capacity Limits: In our technical report, we examine the effects of placing a capacity limit on the number of probes per second that a peer can successfully receive. In many usage scenarios (for example, the default setup presented in Table I), the capacity limit barely comes into play and has a small effect on query satisfaction. This is a promising indication that GUESS is resistant to congestion. When we exacerbate the limited capacity situation by reducing `CacheSize` to be 1% of the network size and using the MFS `QueryProbe` policy, we find that query satisfaction can suffer when nodes are unable to handle incoming load. In light of this potential problem, we experiment with a back-off policy that attempts to reduce the amount of traffic going to overloaded peers.

D. Misbehaving Peers

As with any wide-area P2P system, the existence of malicious peers trying to bring down the network is a possibility that must be considered. As discussed in Section VI-A, if many entries in peers' link caches are dead, then query performance degrades. In the extreme case, the network becomes fragmented. If malicious peers return dead IP addresses, or addresses of other malicious peers, in their Pong messages, they may be able to inject enough bad entries into good peers' caches to bring down the network. Such behavior is called *cache-poisoning*.

In this section, we study the robustness of policies to cache-poisoning. A policy is *robust* if query performance does not significantly degrade as the number of malicious peers in the system increases. To poison good peers' caches, malicious peers will return dead IP addresses in their Pong messages. When probed for a query, malicious peers return no query results; they will only return a corrupt Pong message. The fraction of the network that is malicious is given by `PercentBadPeers`. In our extended report [18], we also study the case in which malicious peers collude by returning the addresses of other malicious peers in their Pong messages. Reference [16] investigates the effect of cache-poisoning on connectivity, and security issues outside of cache-poisoning as well.

Figures 12 and 13 show us query performance in terms of average number of probes per query and satisfaction rate, respectively, as `PercentBadPeers` is varied. The different curves represent different combinations of policy types; the flatter the curve, the more robust the combination. For the experiments shown in these figures, we only vary `QueryProbe`, `QueryPong` and `CacheReplacement` policy types; furthermore, for simplicity, we assume that all three types implement the same policy (e.g., MR/MR/LR, or Ran/Ran/Ran). Each policy has the same qualitative effect regardless of the policy *type* it is used for; therefore the combinations of policy types we consider simply highlight the differences between the policies.

In these figures, we see that the Random and MR policies are robust against cache-poisoning with dead IP addresses, but that MFS is not robust. Although MFS is by far the best-performing policy when all peers are good (`PercentBadPeers` = 0), its performance quickly degrades, reaching a 0% satisfaction rate when 20% of all peers are malicious.

MFS is not robust because it requires that a peer fully trust other peers. If a peer P receives a pong entry reporting that another peer has a particular number of files, P will trust this information, and has no way of proving it wrong. Therefore, all dead IP addresses given by bad peers will be inserted into the link cache. In addition, since bad peers also purport to have many files, bad peers will also be inserted into and remain in the link cache. Though the dead IP addresses will be evicted after a single probe, the malicious peers remaining in the link cache will generate new dead IP addresses in their

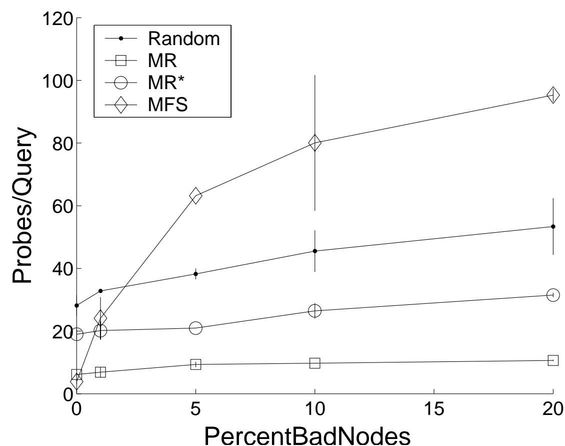

Fig. 12. Average probes per query increases as the number of malicious peers increases

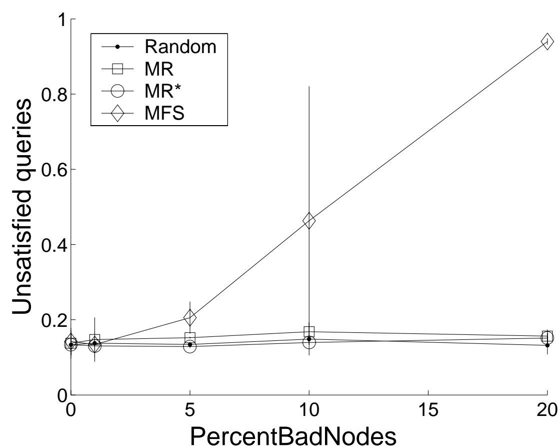

Fig. 13. Satisfaction decreases as the number of malicious peers increases

Pong messages, which will again be inserted into the good peer's link cache.

At first glance, MR should also have poor robustness similar to MFS. Because malicious peers report false `NumRes` values in the Pong message, all the dead IP addresses will be inserted into a good peer's link cache, just as with MFS. However, because the `NumRes` field is reset after each query, the malicious peers in the link cache will have their `NumRes` field set to 0, since they return no results, and will therefore likely be evicted within a short period of time. In contrast, recall that malicious peers remain in the link cache when MFS is used. Because of this difference, MR has excellent robustness.

The Random policy, while robust, is not as efficient as the MR policy, as we saw also in Section VI-B. Therefore, because MR provides the best tradeoff between efficiency and robustness – it has the best performance by far for almost all values of `PercentBadPeers` in the non-collusion scenario – it is the preferred strategy. However, we note that if malicious peers collude, MR is no longer robust. A better policy is MR*, a variation of MR that resets the NumRes field to 0 when an entry is first added to the cache. We refer the reader to [18] for further details in the colluding scenario.

VII. Conclusion

In this paper, we promote the concept of non-forwarding search mechanisms as a viable alternative to popular forwarding-based mechanisms such as Gnutella. Non-forwarding mechanisms, exemplified by the GUESS protocol, can achieve very efficient query performance, but must be carefully deployed. In particular, in this paper we demonstrate how the *policies* used to determine the order of probes, pongs and cache replacement have a dramatic effect on performance and robustness. Through our experiments, we show how the MR policy presents the best tradeoff between efficiency and robustness, and that backoff is required to prevent hotspots from occurring. In the future, we would like to further explore how to make the protocol adapt to changing network conditions, and how to defend against selfish and malicious peers.

References

[1] "KaZaA website," http://www.kazaa.com.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM*, August 2001.

[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM*, August 2001.

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. Middleware 2001*, November 2001.

[5] "Gnutella website," http://www.gnutella.com.

[6] B. Cooper and H. Garcia-Molina, "Ad-hoc, self-supervising peer-to-peer networks," Tech. Rep., Stanford University, 2003.

[7] A. Crespo and H. Garcia-Molina, "Semantic overlay networks," Tech. Rep., Stanford University, 2002.

[8] B. Yang and H. Garcia-Molina, "Improving efficiency of peer-to-peer search," in *Proc. of the 28th ICDCS*, July 2002.

[9] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *Proc. of the 28th ICDCS*, July 2002.

[10] D. Tsoumakos and N. Roussopoulos, "Adaptive probabilistic search for peer-to-peer networks," in *Proc. of the 3rd Conf. on P2P Computing*, September 2003.

[11] B. Yang and H. Garcia-Molina, "Designing a super-peer network," in *Proc. ICDE*, March 2003.

[12] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *Proc. SIGCOMM*, August 2002.

[13] N. Daswani and H. Garcia-Molina, "Query-flood dos attacks in gnutella," in *ACM Conference on Computer and Communications Security*, November 2002.

[14] "GUESS protocol specification," http://groups.yahoo.com/group/-the_gdf/files/Proposals/GUESS/guess_o1.txt.

[15] R. Rodruigues A. Gupta, B. Liskov, "One-hop lookups for peer-to-peer overlays," in *Proc. HotOS*, May 2003.

[16] N. Daswani and H. Garcia-Molina, "Pong-cache poisoning in guess," Tech. Rep., Stanford University, 2003.

[17] S. Saroiu, P. Gummadi, and S. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proc. of the Multimedia Computing and Networking*, January 2002.

[18] B. Yang, P. Vinograd, and H. Garcia-Molina, "Guess: Non-forwarding p2p search mechanism," Tech. Rep., Stanford University, 2003, Available upon request.

[19] B. Yang and H. Garcia-Molina, "Comparing hybrid peer-to-peer systems," in *Proc. of the 27th Intl. Conf. on Very Large Databases*, September 2001.

[20] "LimeWire website," http://www.limewire.com.