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

Beverly Yang Patrick Vinograd Hector Garcia-Molina
{byang, patrickv, hector}@cs.stanford.edu
Computer Science Department, Stanford University

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 result in highly suboptimal performance, and make them susceptible to hotspots and misbehaving peers.

1. 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 [11] 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., [16, 12, 13]) 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 [9], 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 [4, 6], intelligent forwarding of messages within the overlay [19, 5, 17, 14], the use of lightweight indices, data replication [3], and many combinations of the above [2].

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 [5, 19] 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 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 [10] 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 [1]. Like [1], 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 2), based on the specification written by the Gnutella Development Forum [10].
- 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 6.2), but that a naive choice can result in a mechanism that is unfair (Section 6.3), and not robust (Section 6.4).

2. GUESS Protocol

In this section we describe the GUESS protocol for querying and state maintenance. For more details, please refer to the original specification [10]. 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.

2.1. 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 in 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 2.2. Please refer to Figure 1 for an illustration of a GUESS network.

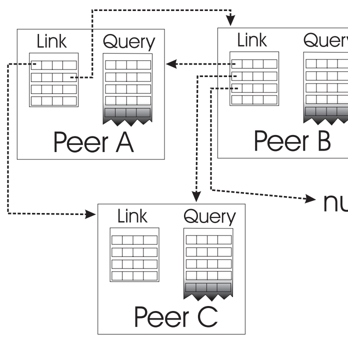

Figure 1: Illustration of a small GUESS network. Note that peer A points to peer C, but C does not point back to A; peer B has one entry pointing to a non-existing peer. Although neighbor pointers do not actually represent open, active connections between peers, they still form a "conceptual" overlay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

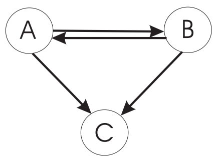

Figure 2: Conceptual overlay representation of the GUESS network in Figure 1

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 2.3.

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 2.2). 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.

2.2. Maintaining State

Because peers in P2P systems typically have short lifetimes [15], 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 sending 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 6.4 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.

2.3. 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 performs 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 a *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.

3. GUESS vs. Gnutella

The Gnutella network, which uses 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 [20].

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 6.2. 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 6.3. 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.

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

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. 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 [7]. 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 [15] 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 6.4.

4. 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.

Most Recently Used (MRU) – prioritizes link 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 very old timestamps are more likely to be dead, resulting in 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

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 1: System and protocol parameters, with default values

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 MFS, 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.

5. 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 1. 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 6, 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 results section, we show how our results scale with network size, when appropriate.

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 [20]. 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 [15]. 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 [8]. 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 [18]. 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 [15] over Gnutella.

A peer sets a maximum number of probes per second

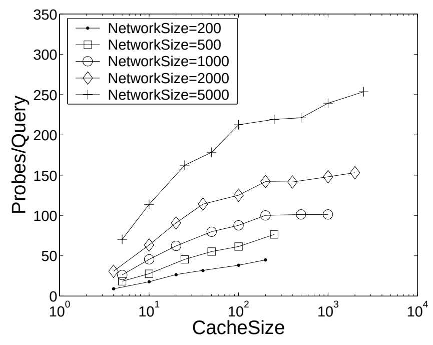

Figure 3: Number of probes increases as cache size increases

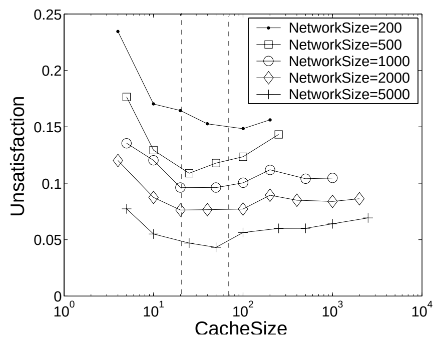

Figure 4: Unsatisfaction experiences a minimum at moderate cache values

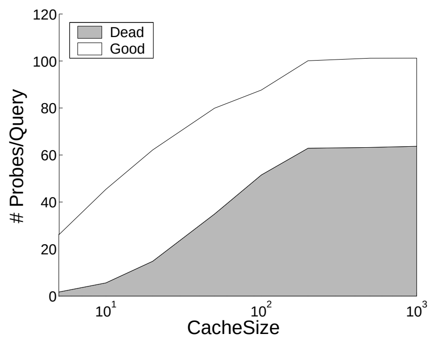

Figure 5: Number of dead probes increases as cache size increases, while good probes experience a maximum at moderate cache value

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 drops queries.

6. 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 6.1) in the face of frequent peer downtimes. We then study the behavior of policies in various system scenarios: in Section 6.2, 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 6.3), as well as the robustness of policies to misbehaving peers (Section 6.4).

6.1. Maintaining the Link Cache

One of the most important issues in maintaining the link cache is selecting the cache size. Intuitively, very small cache sizes result in poor performance, because peers do not have enough “neighbors” to probe. Hence, queries will often not be satisfied. On the other hand, it is not immediately clear whether very large cache sizes will result in good performance, because of the effort necessary to maintain the freshness of the entries.

To investigate the impact of cache size on performance, Figures 3 and 4 look at query performance as cache size is varied, over a number of different network sizes ranging from 200 to 5000 nodes. Due to limitations in our simulator we do not scale beyond 5000. We study cache sizes ranging from 5 (very small) to the size of the network. To provide additional “strain” on the system, we set `LifespanMultiplier` to .2, causing peers to come and go more frequently.

In Figure 3, we see immediately that as the size of the cache grows, regardless of network size, the number of probes per satisfied query also grows. A reasonable explanation for this phenomenon immediately comes to mind: as the size of the cache increases, there are more peers to probe. Hence, many of the queries that were previously unsatisfied will now be satisfied, albeit at a higher number of probes. The consequence of this explanation is that as cache size increases, the unsatisfaction rate decreases.

Surprisingly, we see in Figure 4 that this explanation does not hold. As we expected, unsatisfaction is high when cache size is very small. However, across all network sizes shown, unsatisfaction experiences a minimum at a moderate cache size, after which point it increases once more. In other words, very large cache size not only results in more expensive queries, it also results in lower satisfaction.

To understand this phenomenon, in Figure 5 we take a closer look at the number of probes required to answer a query. For clarity, we look only at our default network size of 1000. The probes are broken down into “good” and “dead” probes: a “good” probe is a probe sent to a live peer, while a “dead” probe is sent to a peer that is no longer online. Good and dead probes are represented in Figure 5 by the white and gray regions, respectively. We see from Figure 5 that as cache size increases, the number of *dead* probes increases dramatically at first, and then levels off. The number of good probes, however, does *not* increase. Hence, although larger cache sizes result in a larger number of probes, they do not translate to more satisfied queries, because the additional probes are all useless.

In fact, the number of good probes experiences a maximum at a cache size of 20. At this maximum, the number of good probes is almost 30% higher than when cache size is 200. Since the satisfaction of a query depends only on the number of good probes, we can understand why satisfaction also reaches a maximum at a cache size of 20.

At larger cache sizes, the link cache maintenance effort

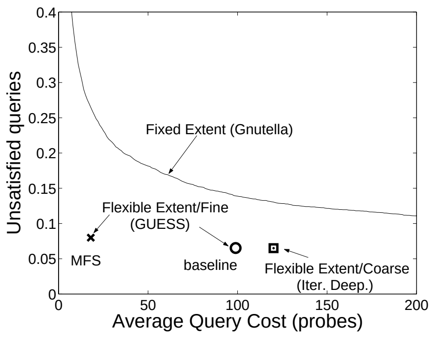

Figure 6: For a given average query cost, unsatisfaction rate is lowest with a fine-grained flexible extent provided by GUESS

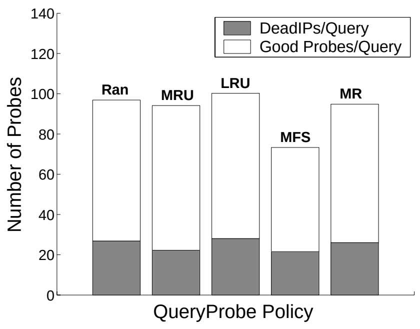

Figure 7: Probes/Query for different QueryProbe policies

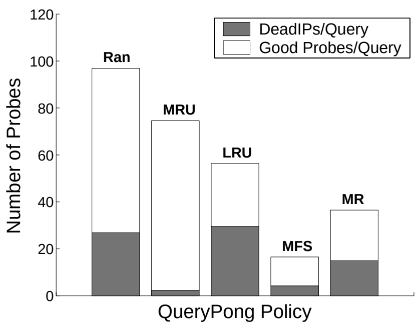

Figure 8: Probes/Query for different QueryPong policies

is "spread too thin" over a large number of entries; hence, the ratio of live to dead cache entries decreases. As a result, a larger fraction of probes are dead during query execution. In addition, because Pong messages contain a sample of link cache entries, the ratio of live to dead *query cache* entries will also decrease as link cache size increases.

In terms of how to select a cache size, clearly, moderate cache sizes have the best cost/performance tradeoff. In fact, in Figure 4, we see that a cache size in the range of 20-70 (range marked by dotted lines) results in best satisfaction – for all network sizes studied, the optimal cache size does not change with network size. Our experiments show strong evidence that optimal cache size grows very slowly, if at all. Hence, even with very large network sizes, a reasonably small cache size is sufficient for good performance.

6.2. 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 (e.g., Gnutella), a coarse-grained flexible extent mechanism (e.g., the iterative deepening [19] approach), and a fine-grained flexible extent mechanism (e.g., GUESS). With regards to extent, iterative deepening (shown by a square in Figure 6) is conceptually similar to GUESS, except that many peers (e.g., hundreds) are probed in each iteration, instead of just one. The 'o' and 'x' mark the points in the figure that represents GUESS, using the Random baseline policy and `QueryPong = MFS`, respectively. 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.

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 with the baseline policy, and 8% unsatisfaction with an average of 17 probes with `Query-Pong = MFS`. In contrast, a fixed extent mechanism such as Gnutella would require 1000 probes for 6% unsatisfaction, and 540 probes for 8% unsatisfaction – well over an order of magnitude higher than GUESS. Iterative deepening is also less efficient than GUESS, given that its control over extent is coarse-grained; however, we see even limited control of extent can result in a fairly good balance between cost and quality.

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 more 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. In the remainder of this section, we will focus on the challenges and opportunities afforded by a flexible extent: response time, and policy selection for efficiency.

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 1. 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 obser-

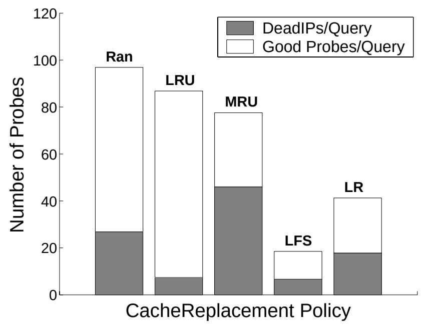

Figure 9: **Probes/Query for different CacheReplacement policies**

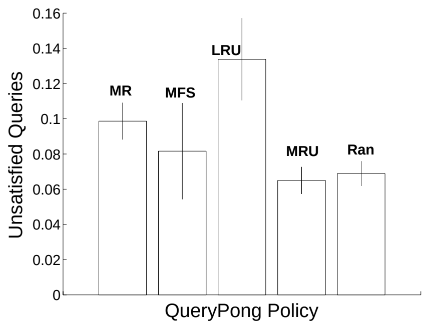

Figure 10: **Percentage of queries that are not satisfied, for different `QueryPong` policies**

vation 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.

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.

Response Time: Because each probe is performed sequentially, the response time of GUESS is linear in the number of probes required. Therefore, selecting a good policy is critical not only for efficiency, but for user experience as well.

To improve response time, a peer may send out k probes in parallel. Doing so will only increase the required number of probes by at most $k - 1$, but will decrease response time by a factor of k – a good tradeoff for any moderate value of k (e.g., 10). For example, when `QueryPong = MFS`, the average number of probes required is 17. If we set $k = 5$, and each set of parallel probes is sent out every .2 seconds (according to the GUESS specification [10]), then the average number of probes is at most 21, meaning average response time is less than 1 second.

Of course, though average response time may be low, worst-case response time is still bad. If 1000 probes are required to satisfy a query, then using the parameters in the previous example, 50 seconds are required to answer the query. A more sophisticated solution may adaptively increase k if successive sets of parallel probes are unsuccessful. We leave such a technique to future work.

6.3. Individual Loads

One of the main problems encountered by the original Gnutella protocol was congestion; therefore, it makes sense to examine whether GUESS might be susceptible to such problems as well. While GUESS does not cause queries to be magnified by a flooding mechanism (as in Gnutella), 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 for some peers, and then examine possible ways to remedy such a condition.

Fairness: Figure 11 shows the peers from a simulation run, ranked by number of probes received during their lifetimes, for different combinations of `QueryProbe` and `CacheReplacement` policies. The rank is shown on a

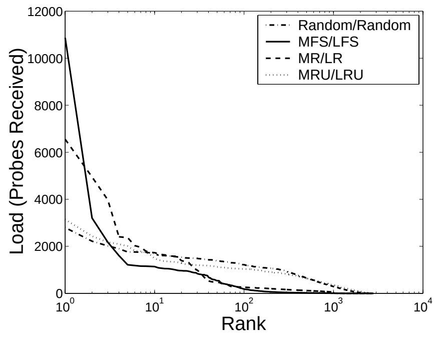

Figure 11: Ranked distribution of load (probes received) for different combinations of `QueryProbe` and `CacheReplacement` policies

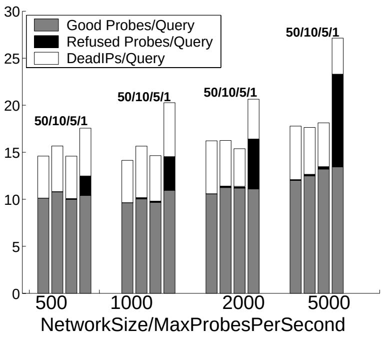

Figure 12: For large networks, limited capacity leads to more refused probes

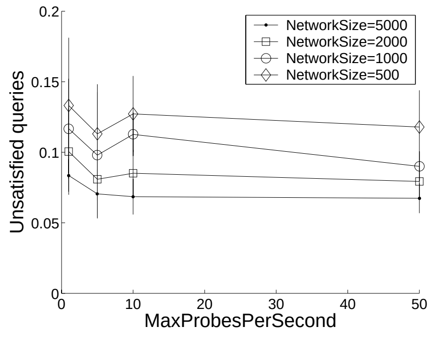

Figure 13: Query satisfaction is not affected by the capacity limits, even when a significant number of probes are refused

logarithmic scale so that the highest-loaded peers are visible. We see that for MFS/LFS, and MR/LR, the load is weighted heavily to a small number of peers who do most of the work. On the other hand the curve for the Random/Random policy is much more level, meaning that the load is spread more evenly across all peers.

Despite the fact that we can choose a more fair policy, it is not clear that we want to. While the load is spread more evenly with the Random/Random policy, the total number of probes sent is over 8 times as many as when MFS/LFS is used. The probes are received in a more fair manner, but by peers who are unable to satisfy queries. This wastes both time and bandwidth for the querying peers as well as the receiving peers. So, fairness may be trumped by other concerns such as overall efficiency. However, this example does illustrate that some peers may encounter very high loads. Therefore, we should determine how the system will react if the capacity of these peers is exceeded.

Capacity Limits: In order to simulate peers having limited capacities we introduce the parameter `MaxProbesPerSecond`, representing the maximum number of probes that a peer can process within a one-second window. Any probes received beyond this limit will be dropped. In the following figures, we refer to these dropped probes as "refused" probes. Although different peers may have different capacities, we assume that in the interest of fairness, all peers set the same limit.

Under our default scenario, using a `NetworkSize` of 1000, a `CacheSize` of 100, and the default `QueryProbe` policy of Random, limited capacity barely comes into play. Any value of `MaxProbesPerSecond` greater than 1 results in less than 1 probe per query being refused (and in most cases, zero refused probes). Even with a capacity of just 1 probe per second, the number of unsatisfied queries increases by less than 2 percent. This is a promising sign that GUESS is resistant to congestion, which is one of the goals of the protocol.

We can see the effects of a limited capacity more clearly in larger networks, and in policies that are less fair, such as MFS and MR. Figure 12 shows the number of probes per query for different capacities and network sizes, using the MR policies. For example, the left-most group of bars in the figure shows the average number of probes per query for a network of 500 nodes, with capacity decreasing from left to right. As the network grows, the number of good probes and probes to dead peers remain pretty steady. However, the number of refused probes increases with network size. There are a few nodes that consistently provide good results and thus reside in many link caches, and they become overloaded by the large number of peers who select them as probe recipients.

Despite the increase in refused probes, however, we see in Figure 13 that query satisfaction is hardly affected at all. In our experimentation, despite some of the peers becoming overloaded at times, there were enough other peers in the network that were capable of satisfying queries, and so satisfaction rates did not decrease. As the network becomes larger, there are more peers sending probes, but there are also more peers to service queries, so the network may be self-sufficient.

The GUESS policy contains an inherent throttling mechanism for overloaded peers which helps in the face of limited capacity. When a peer gets overloaded, and drops the probes it cannot handle, the probing nodes will then remove the overloaded peer from their caches (believing it is dead). By removing an entry from its cache, a peer will will not propagate the entry in its Pong messages, which in turn reduces the number of probes that the overloaded node might receive in the near future. A nice feature of this mechanism is that in GUESS, the lack of response from one overloaded peer does not unduly interfere with the overall query, whereas in Gnutella, a drop might prevent hundreds of other potential recipients from seeing the query.

Nevertheless, GUESS cannot get around the fact that there are only a small number of peers that tend to share a large number of files. If a popular file is only located at a few overloaded nodes, many queries will still go unsatisfied. Ultimately, having a high degree of content replication is the best solution for allowing a file sharing network to scale successfully; it allows many querying peers to locate content without placing a huge load on any one or any handful of nodes. In a public trading network like Gnutella, where it is difficult to require people to provide content, a better solution might be to provide incentives for sharing and replicating content.

6.4. Misbehaving Peers

When designing any wide-area P2P system, one must consider the possibility of malicious peers. The GUESS protocol is particularly vulnerable to the *cache-poisoning* attack, where malicious peers inject IP addresses of dead or malicious peers into good peers' caches via corrupt Pong messages. If cache-poisoning is widespread enough, it can degrade performance significantly.

In our extended report [20], we study the robustness of policies against cache poisoning. A policy is considered *robust* if its performance does not significantly degrade as the percentage of malicious peers in the system increases. From our investigation, we found that when malicious peers do not collude, the MR policy has the best combination of performance and robustness. Only if a very small percentage of peers are malicious does MFS outperform MR. When malicious peers collude, however, MR is no longer robust. Instead, a more robust policy is MR*, a variation of MR where peers reset the NumRes field to 0 when an entry is first added to the cache. Because peers no longer rely on information provided by other peers, MR* has good robustness, like Random, and it outperforms Random in terms of number of probes per query and unsatisfaction rate.

Although MR and MFS are not inherently robust to malicious collusions, they can still be practical given an effective means to detect and react against malicious behavior [8]. For further details on our experiments and a discussion on cache-poisoning, we refer readers to our extended report [20].

7. 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. From our experiments, we conclude that the MR policy presents the best tradeoff between efficiency and robustness, while scaling fairly well with network size. Therefore, our recommendation for a first-generation implementation of GUESS would be to use the MR policy. 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] R. Rodruigues A. Gupta, B. Liskov. One-hop lookups for peer-to-peer overlays. In *Proc. HotOS*, May 2003.

[2] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proc. of SIGCOMM*, August 2003.

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

[4] B. Cooper and H. Garcia-Molina. Ad-hoc, self-supervising peer-to-peer networks. Technical report, Stanford University, 2003.

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

[6] A. Crespo and H. Garcia-Molina. Semantic overlay networks. Technical report, Stanford University, 2002.

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

[8] N. Daswani and H. Garcia-Molina. Pong-cache poisoning in guess. Technical report, Stanford University, 2003.

[9] Gnutella website. http://www.gnutella.com.

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

[11] KaZaA website. http://www.kazaa.com.

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

[13] 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.

[14] L. Guo S. Jiang and X. Zhang. Lightflood: an efficient flooding scheme for file search in unstructured peer-to-peer systems. In *Proc. of 2003 Intl. Conf. on Parallel Processing*.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] B. Yang, P. Vinograd, and H. Garcia-Molina. Guess: Non-forwarding p2p search mechanism. Technical report, Stanford University, 2003. Available at http://dbpubs.stanford.edu/pub/2003-73.