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 [13] 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., [19, 15, 16]) 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 [10], 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 [22, 5, 20, 17], 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, 22] 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 [11] 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 [11].
- 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 [11]. 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

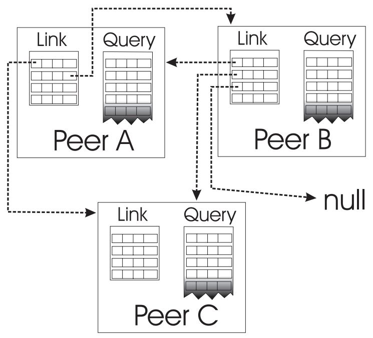

Figure 1: **Illustration of a small GUESS network. Note that peer A points to peer C, but C does not point back to A; peer B has one entry pointing to a non-existing peer. Although neighbor pointers do not actually represent open, active connections between peers, they still form a "conceptual" overlay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

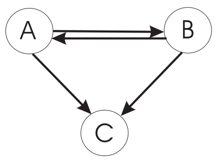

Figure 2: **Conceptual overlay representation of the GUESS network in Figure 1**

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.

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:

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

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 [18], 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

To understand the advantages and drawbacks of the non-forwarding GUESS protocol at a high level, we would like to compare it to Gnutella, the predecessor to GUESS. Gnutella is a forwarding-based protocol, and is also the closest existing protocol to GUESS in terms of functionality and intended use. Unfortunately, a direct comparison between GUESS and Gnutella is difficult because a great deal of research has been done to improve Gnutella. GUESS on the other hand is relatively new and there are many possible improvements that might be made. Furthermore, it is not feasible to implement all of the possible optimizations for each protocol to produce a quantitative comparison, nor is it reasonable to compare implementations of the basic versions of the protocols which have some obvious shortcomings. Hence, in this section we present a high-level, *qualitative* comparison of the efficiency and security characteristics of the two protocols. We attempt to focus on the fundamental characteristics of GUESS and Gnutella, which will hold true despite optimizations on either protocol. In Section 6, we perform an in-depth quantitative analysis of the GUESS protocol only.

3.1. Query Performance

Perhaps the primary point of comparison between GUESS and Gnutella is that of efficiency of queries, in

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

terms of network resources used. In Gnutella, the location and extent of a query are determined by the overlay topology; queries will be received by whichever peers happen to be within a certain radius of the originator. Because the execution of a query is not adaptive, it is often inefficient. For queries for popular items, Gnutella returns far too many results; for queries for rare items, Gnutella does not return enough results. Furthermore, although different peers may have different needs (e.g., search for different types of content), the basic Gnutella protocol does not differentiate between peers.

In contrast, the GUESS search protocol provides each client with fine-grained control over the execution of the query. In particular, peers can control: (1) the **order** in which other peers are probed, and (2) the **extent** of a query. By controlling the order of probes, a peer in GUESS can use metadata or past experience to identify other peers that can quickly return results. In Section 6.2 we will see how the order in which peers are probed has an enormous impact on query performance. One drawback to ordering probes is the possibility of hotspots, where many peers all try to probe the same productive peer, and that productive peer is overloaded. This issue is further explored in Section 6.3. We also note that to a limited extent, techniques such as *routing indices* [5] and *directed breadth-first-search* [22] allow peers in Gnutella to control the order in which peers receive their query.

By probing peers iteratively (rather than all in parallel), a peer can adapting the extent of a query to the popularity of the item being sought, thereby resulting in fewer wasted probes, and better satisfaction results. If a file is popular and adequate results are obtained quickly, a search can be terminated after querying a small number of peers. Alternatively, if a file is rare, the search can continue over a much larger number of peers until the search is satisfied. Again, to a limited degree, techniques such as *iterative deepening* [22] allow peers to control the extent of the query. However, the degree of control is much more coarse-grained than in GUESS, thereby resulting in greater inefficiency.

One important drawback to probing peers iteratively is poor response time. For example, if 1000 peers are probed before the query is satisfied, and peers must wait .2 seconds between successive probes, the query will be satisfied only after 200 seconds – possibly too long a time to satisfy the user. In contrast, because all peers in Gnutella are probed in parallel (more or less), response time for Gnutella is very good. Response time in GUESS may be improved by parallel walks (probing a small number of peers in parallel), or adapting the rate of probes according to how many peers have already been probed. For example, for every 10 seconds that pass by without finding a result, the rate of probes may be doubled. Furthermore, if the order in which peers are probed is carefully chosen, response time may actually be better in GUESS than Gnutella, in some cases. Again, response time in GUESS is further explored in Section 6.

3.2. State Maintenance

Although GUESS is generally superior to Gnutella in terms of query efficiency, it is inferior to Gnutella in regards to the cost and complexity of state maintenance. A Gnutella peer maintains a small number of open network connections to its neighbors in the overlay network. In contrast, a GUESS peer must maintain pointers in a fairly large pong cache. Although the size of state is generally not an issue given the availability and low cost of memory, the amount of network traffic needed to maintain the cache can be high if the cache is large (which is generally true if the network is large). In Section 6.1 we investigate the cost of maintaining state in greater detail.

Related to the above concept of state is the consistency of the network over time. In Gnutella, because state is localized to the immediate neighbors of each peer, the process of joining and leaving the network is simple. A peer notifies its neighbors when it joins or leaves the network, and only these neighbors need to modify their state to maintain a consistent view of the network. In contrast, a peer in GUESS may belong in the link caches of many other peers, but because neighbor relationships are 1-way, does not have knowledge of who these other peers are. Hence, when a peer leaves the network, it leaves without notifying anyone. Other peers must then discover the death of that peer with an unsuccessful probe, which results in wasted work. Furthermore, when a peer joins the network, it must actively introduce itself into other peers' pong caches, before it can be of any use to the network.

3.3. Security

Because peers in a large-scale P2P system are autonomous and may come from competing organizations, a good search protocol must be robust against misbehaving peers. We define two types of misbehavior: *selfish* behavior, and *malicious* behavior.

Selfish Peers. A selfish peer tries to "game" the system in order to maximize its own utility, regardless of the cost it imposes on other peers. It does not desire to bring down the system, since that would cause poor utility to the selfish peer as well. With a search protocol like GUESS or Gnutella, a selfish peer will attempt to maximize the number of results it receives, while minimizing response time.

In Gnutella, it is difficult for a single peer to game the system, because peers do not have control over the execution of their own query. If a peer wishes to receive more results for its queries in less time, then it must connect to more neighbors. Having more neighbors increases the load on that peer, due to the forwarding traffic in Gnutella; hence, in order to improve its search performance, a peer must first contribute greater resources to the system. Although a peer may drop messages in order to reduce its load, it is a simple matter for its neighbors to observe that they rarely receive traffic (e.g., query response messages) from the selfish peer, and to disconnect from that peer in return.

In GUESS, it is very easy for a peer to game the system, because peers have complete control over the execution of their own queries. Rather than iteratively probe peers on a query, a selfish peer can simply probe thousands of peers at a time. This behavior may incur a much higher load on other peers (e.g., if the query could have been answered with 20 probes), while possibly drastically improving the response time of the search for the querying peer. If all peers act according to their best interests, the system might fail as if under a denial of service (DoS) attack. Therefore, in order for GUESS to work, there must be a way to give peers an incentive to adhere to the protocol. One straightforward proposal is to have peers "pay" for each probe. Peers will then be motivated to probe as few peers as possible to answer their queries. Such a solution does require a payment mechanism, such as [23]. Other ways to limit selfish behavior remains an open problem.

Malicious Peers. The goal of a malicious peer is to make the system unusable. For the purposes of comparison, we will focus on attacks on the protocol itself; we ignore attacks such that are unrelated to the choice of protocol, such as returning inauthentic documents. We will discuss malicious behavior further in Section 6.4, where we will also consider ways to improve the robustness of the GUESS protocol.

The Gnutella protocol is susceptible to application-level *denial-of-service* (DoS) attacks, due to the "amplification effect" inherent to the protocol. Because queries are flooded across the network, a malicious peer needs to expend little effort to cause a much higher load on the system. Reference [8] presents some techniques for preventing DoS attacks in Gnutella, though naturally, these preventative measures result in lower efficiency in the case of cooperative behavior.

Some studies (e.g., [18]) also suggest that Gnutella is susceptible to *fragmentation* attacks, where highly-connected peers are brought down via network-level DoS attacks, and the overlay topology is fragmented. Fragmentation is bad because it can result in worse quality of response (e.g., fewer responses) for queries. However, this weakness is not inherent to the Gnutella protocol itself, but is an artifact of the type of topology (power-law) that naturally arises from peers' local connection decisions [[REFERENCE?]]. The Gnutella network can be made more robust, for example, by imposing simple limits on the number of connections peers are allowed to make.

The GUESS protocol is not as susceptible to application-level DoS attacks, because query messages are not amplified – the malicious peer will need to expend effort proportional to the number of messages sent. However, as discussed earlier, peers acting selfishly can induce a similar effect.

On the other hand, the GUESS protocol is susceptible to fragmentation attacks, if malicious peers collude. It is fairly easy for a malicious peer to insert itself or other malicious peers into good peers' link caches, via Pong messages. If a large percentage of many peers' link caches are filled with malicious peers, and all malicious peers disappear from the system at the same time, then the conceptual overlay formed by peers' link caches will become fragmented. Such behavior by malicious peers is known as "cache poisoning". The effects of cache poisoning and prevention are discussed further in Section 6.4.

Finally, we note that other security-related issues, such as privacy and anonymity, are easier to achieve in Gnutella than in GUESS. Because peers control their own queries in GUESS, there is no way for a peer to hide its query behavior from other observing peers. Therefore, GUESS may not be appropriate for certain privacy-sensitive applications.

In summary, giving peers control over the execution of their query, while potentially improving efficiency of the system (as discussed in the previous section), also opens up the possibility of abuse and manipulation. If we are to give such control to peers, we must (1) learn how peers can best use this control to optimize query performance, (2) add other mechanisms or incentives to enforce correct behavior, and (3) learn how peers can avoid being manipulated by malicious peers. For the remainder of this paper, we focus on the first question of performance optimization, but address the second and third questions as well. (weak)

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 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 Tables 1 and 2, respectively. 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.

5.1. Simulation Framework

In a given simulation run, we simulate `NetworkSize` peers participating in the GUESS protocol. At time 0, all peers are alive. Each link cache is seeded with `CacheSeedSize` living peers. We found that as long as `CacheSeedSize` was small (e.g., approximately `NetworkSize`/100), the value of `CacheSeedSize` did not affect performance results.

As time progresses, peers will die. A large sample of peer lifetimes in the Gnutella network was measured in [18]. 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.

Name	Default	Description
NetworkSize	1000	Number of peers in the network
NumDesiredResults	1	Number of desired results per query
LifespanMultiplier	1	Parameter used to vary peer lifespans
Query Rate	$9.26 \cdot 10^{-3}$	The expected number of queries per user per second
MaxProbesPerSecond	100	Maximum number of probes per second a peer is willing or able to handle
PercentBadPeers	0	The percentage of peers in the network that are malicious
BadPongBehavior	Dead	The type of IP address a bad peer returns in a pong (Dead, Bad, or Good)

Table 1: System parameters, and default values

Name	Default	Description
QueryProbe	Random	The policy used to determine which peer to probe for a query
QueryPong	Random	The policy used to determine which IP addresses to return in a pong
PingProbe	Random	The policy used to determine which peer to probe for a ping
PingPong	Random	The policy used to determine which IP addresses to return in a pong
CacheReplacement	Random	The policy used to determine which peer to evict from the link cache
PingInterval	30s	Elapsed time between pings
CacheSize	100	Size of the link cache
ResetNumResults	No	Flag indicating whether to reset the responses field in a cache entry
DoBackoff	No	Flag indicating whether to perform backoff
PongSize	5	Number of IP addresses per pong
IntroProb	.1	Probability of adding to your cache a peer who probes you.

Table 2: Protocol parameters, and default values

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 [9]. 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. A friend can be the IP address of a peer that the new peer saved from when it was last online, or peers can obtain the IP address of a friend from a "pong server." A pong server (e.g., LimeWire [14] for the Gnutella network) is a service that keeps track of live peers in the system. Because such a service is centralized and potentially expensive (e.g., if it receives many requests), we do not wish to make heavy use of the service. In particular, it is not reasonable for each new peer to ask the pong server to initialize its link cache with the IP addresses of many live peers.

The generation of queries at each peer follows a bursty pattern, in which a number of queries (number uniformly chosen between 1 and 5) are submitted in succession, followed by a long wait. The arrival of bursts follow a Poisson process, and the overall rate of queries per user is given by `Query Rate`. When a peer has a query, it will iteratively probe the peers in its link cache, which is of size `CacheSize`, until `NumDesiredResults` results are returned. The order in which the peer probes other peers is determined by the `QueryProbe` policy. For the purposes of our simulation we assume that timeout period between probes is long enough for a live peer to respond within the period. In real life some peers may take longer than the timeout period; however, this would simply mean that a few unnecessary probes are performed, and would not significantly alter our results.

When a peer is probed, and it is not *overloaded* (described below), it will see if any files in its collection answer the query. To determine whether a peer returns a result for a query, we use the query model developed in [21]. 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 a peer returning a result depends partially on the number of files owned by that peer; number of files owned by peers are assigned according to the distribution of files measured by [18] over Gnutella. Whether or not a match is found for the query, the responding peer returns a Pong message with `PongSize` IP addresses to the querying peer. The IP addresses are selected from the link cache according to the `QueryPong` policy. When the querying peer receives the Pong, it will possibly insert the new IP addresses into its link cache, according to the `CacheReplacement` policy. The peer will also add the IP addresses to its query cache, if they have not already been seen before.

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 6.3.

Peers maintain the IP addresses in their link cache by pinging one peer every `PingInterval` seconds. The order in which peers are pinged is determined by the `PingProbe` policy. As with query probes, when a peer receives a ping, it will respond with a Pong message containing `PongSize` IP addresses. The IP addresses are selected from the link cache according to the `PingPong` policy. When the querying peer receives the Pong, it will possibly insert the new IP addresses into its link cache, according to the `CacheReplacement` policy.

The `IntroduceProb` parameter sets the probability of adding a peer to your cache when that peer initiates an interaction, according to the introduction policy we use (discussion of introduction policies is found in Section 2.2). The parameters `PercentBadPeers` and `BadPongBehavior` describe malicious peer behavior, and are discussed further in Section 6.4.

When a peer is probed, to determine whether it returns a result for the query, we use the query model developed in [21]. 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 [18] over Gnutella.

A peer sets 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 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

Two of the most important issues in maintaining the link cache is the size of the cache, and the rate at which the cache is maintained via Pings.

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.

Cache Size	Fraction Live	Absolute Live
10	.822	8.0
20	.759	14.8
50	.605	28.5
100	.418	36.2
200	.330	41.9
500	.309	41.9

Table 3: Breakdown of live cache entries for varying PingIntervals

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. 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.

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, however, 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, Table 3 looks at the average fraction of cache entries that are live, as well as the average absolute number of live cache entries, for various cache sizes (using the default NetworkSize = 1000). Note that the total number of cache entries may be less than the cache size itself. The CacheSize parameter only tells us the capacity of the cache. Often caches are not full because entries are evicted if they are found to be dead.

Let us first consider the fraction of live cache entries (column 2) in Table 3. Here we see that the fraction of live entries greatly decreases as the size of the cache grows (although absolute number of live entries increases). The fraction of live entries is important because, given a Random probe policy, it roughly governs the fraction of probes to live peers (i.e., "good" probes), and the fraction of probes to dead peers (i.e., "dead" probes).

The effect of this ratio can be seen in Figure 5, where we take a closer look at the number of probes required to

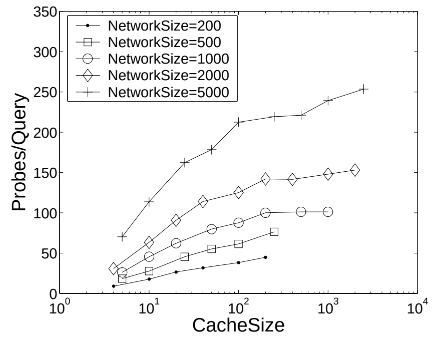

Figure 3: **Number of probes increases as cache size increases**

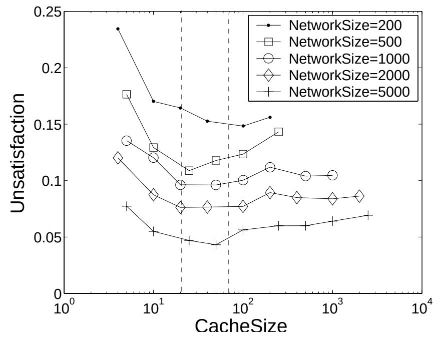

Figure 4: **Unsatisfaction experiences a minimum at moderate cache values**

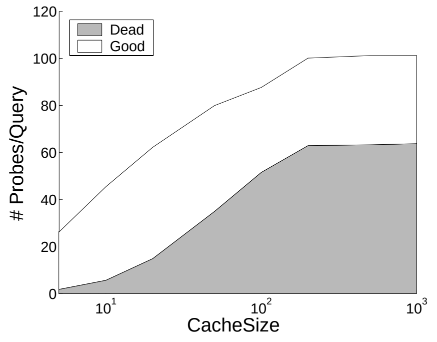

Figure 5: **Number of dead probes increases as cache size increases, while good probes experience a maximum at moderate cache value**

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. 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 local 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 (Figure 4). At larger cache sizes, the link cache maintainence effort is "spread too thin" over a large number of entries; hence, fewer entries are good when a query is executed. In addition, because Pong messages will contain more dead IPs if peers' link caches contain more dead IPs, the total number of useful *query cache* entries will also decrease proportionally.

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. Intuitively, optimal cache size might grow with network size, and experiments with very large network sizes (e.g., a million) should make this apparent. However, 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.

Ping Interval. A fragmented overlay is bad because it degrades query performance, and unless there is some form of centralized boot-strapping server (e.g., pong servers such as those run by LimeWire [14] for Gnutella), the network is unlikely to heal. Here, we study how PingInterval must be set in order to maintain a *connected* topology. All cache sizes considered in our study thus far are large enough to maintain a connected topology, assuming cache entries are well-maintained. In order to keep a well-maintained cache, PingInterval may need to be adjusted as CacheSize and NetworkSize vary.

We note that cache is maintained via Ping messages, and also via query messages, since peers respond to queries with both query responses and Pong messages. To isolate the effect of Pings, in the remainder of the section we do not simulate queries.

Figure 6 shows us the size of the largest connected component in a GUESS "conceptual overlay" for various cache sizes, as PingInterval varies across the x-axis. A fully connected network will have the largest connected component of size 1000. As expected, the smaller the PingInterval, the more connected the overlay is. As PingInterval increases, the overlay begins to fragment.

Comparing across the curves in this figure, we see that the smaller the cache size is, the smaller PingInterval must be. That is, smaller cache sizes must be more closely maintained. At first this result may seem surprising when compared to our results in the previous section, where smaller cache sizes result in better query performance. The reason for this difference is that for query performance, the *ratio* of live to dead peers is critical, whereas for connectivity, the *absolute number* of live peers is more important. Returning to Table 3, we see that although smaller cache sizes can have a dramatically higher fraction of live entries than large cache sizes, the absolute number of live cache entries is larger for large cache sizes.

In terms of how NetworkSize affects PingInterval, we found that for moderate cache sizes (e.g., between 20 and 100), the connectivity of the overlay is largely unaffected by network size. Figure 7 shows the relatively connectivity of the GUESS overlay as PingInterval varies along the x-axis, for various network sizes. In this figure, CacheSize = 20,

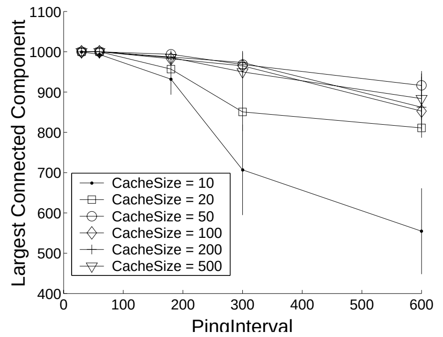

Figure 6: Small cache sizes are most negatively affected by long ping intervals

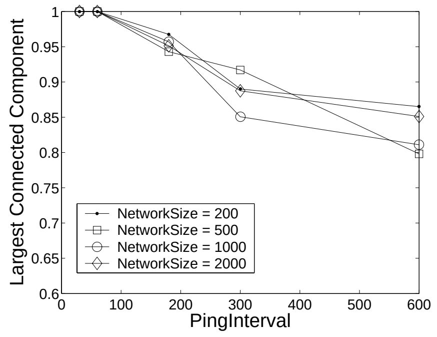

Figure 7: Selection of ping interval is largely independent of network size

regardless of NetworkSize. We see from Figure 7 that the relative size of the largest connected component is roughly the same across network sizes, for a given PingInterval. Although one might expect that larger network sizes require more links per peer to remain connected, this is in fact not the case; we have experimental evidence showing that each node in a random graph requires a small constant number of links for the graph to be connected with high probability.

In conclusion, there does not appear to be a set rule for good PingIntervals, but this parameter may be adjusted at runtime. After selecting a cache size that results in good query performance, a peer should adjust its PingInterval to maintain a certain threshhold of live entries in its cache. While sending query or Ping messages, if a peer discovers that many of its probes are to dead addresses, the peer should decrease its PingInterval. On the other hand, if a peer discovers that almost all its entries are live, then it may increase its PingInterval, if the current overhead of Ping/Pong messages is placing a heavy burden on its resources (which is possible, though unlikely). Given these guidelines, along with the guidelines for selecting a cache size, peers can maintain the health of the network with a very reasonable overhead.

6.2. Basic Policies

Flexible Extent: First, we highlight the performance benefits of a completely flexible query extent. Figure 8 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 [22] approach), and a fine-grained flexible extent mechanism (e.g., GUESS). With regards to extent, iterative deepening (shown by a square in Figure 8) 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 8, 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 `QueryPong = 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

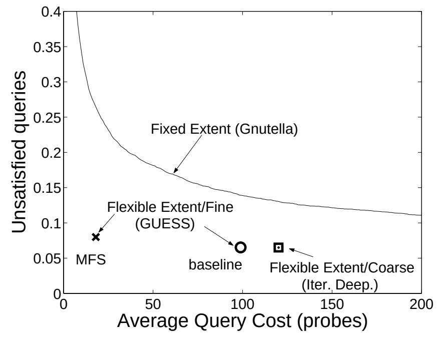

Figure 8: **For a given average query cost, unsatisfaction rate is lowest with a fine-grained flexible extent provided by GUESS**

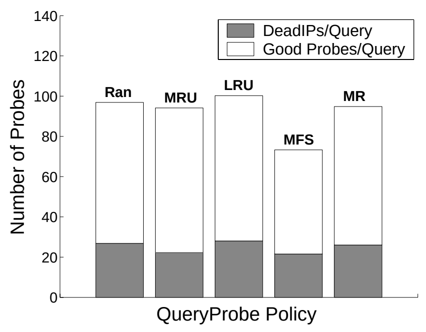

Figure 9: **Probes/Query for different QueryProbe policies**

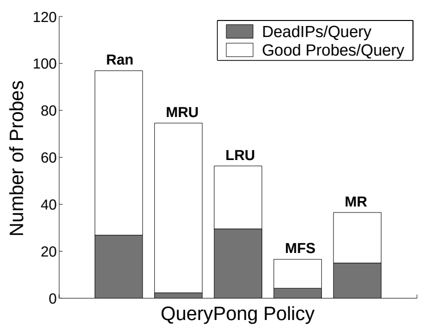

Figure 10: **Probes/Query for different QueryPong policies**

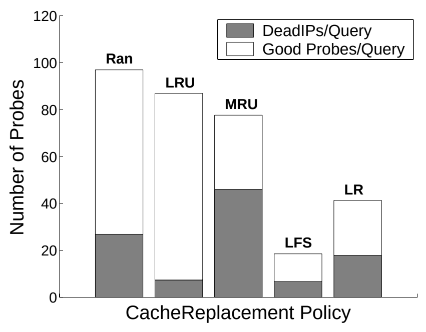

Figure 11: **Probes/Query for different CacheReplacement policies**

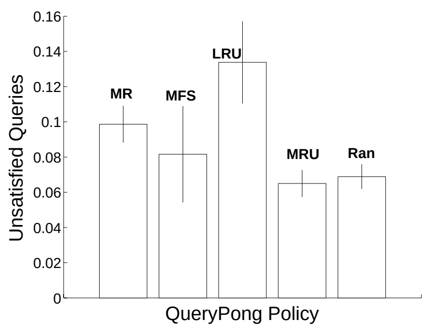

Figure 12: **Percentage of queries that are not satisfied, for different `QueryPong` policies**

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 9, 10, and 11, our immediate observation is that choosing different policies can have a dramatic impact on performance. For example, in Figure 10 we see that changing the `QueryPong` policy can reduce cost (Probes/Query) by a factor of four. Likewise, in Figure 11 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 11, 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 10 and Figure 11). 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 12, 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 [11]), 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 13 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 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 14 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 15 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 over-

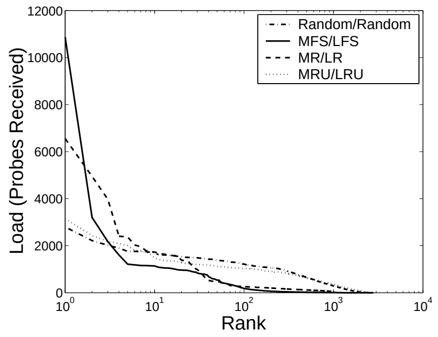

Figure 13: Ranked distribution of load (probes received) for different combinations of `QueryProbe` and `CacheReplacement` policies

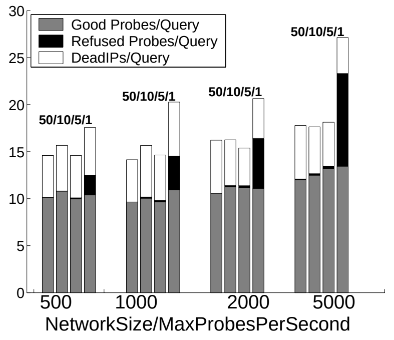

Figure 14: For large networks, limited capacity leads to more refused probes

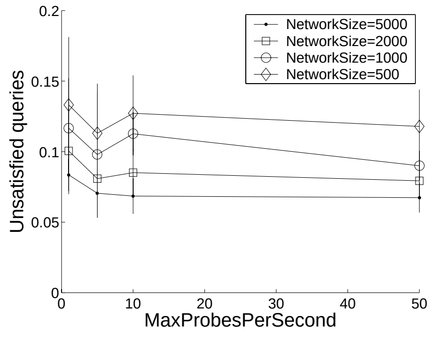

Figure 15: Query satisfaction is not affected by the capacity limits, even when a significant number of probes are refused

loaded 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. Malicious Attacks

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. In most cases, the goal of a malicious peer is to make the system unusable. There are two main ways to accomplish this goal. First, the malicious peer may distribute corrupt on inauthentic files. If many peers serve bad files, then good users may eventually become frustrated and leave. A second way to undermine the system is to attack weaknesses in the protocol itself. In Section 3 we already discuss how it is easy to flood peers in the GUESS protocol, and that some form of payment or score management is required to counter flooding. Another way to attack the GUESS protocol is to "poison" the link caches of good peers. As discussed in Section 6.1, 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.

While the issue of document authenticity and reputation is important, it has been studied in various places (e.g., [12, 7]), and many techniques developed can be used on top of the GUESS protocol. Likewise, a fairly straightforward solution to flooding is available given the existence of an efficient score or payment mechanism. Hence, 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.

In addition to the five policies presented in Section 6.2, we also look at a sixth policy: MR*. As with MR, a peer P using MR* orders entries according to the number of results returned in the past, as specified by the `NumRes` field of the cache entry. Unlike MR, however, the MR* policy ignores the value of `NumRes` (resetting it to 0) if the field was set by a different peer. As a result, peer P will order entries based solely on P's direct experience with those peers, rather than the history of experience that other peers have had. While this policy causes potentially useful information to be lost, it also avoids making bad decisions based on corrupt information.

To poison good peers' caches, malicious peers will return either dead IP addresses in their Pong messages (`BadPongBehavior` = Dead), or they will return IP addresses of other malicious peers (`BadPongBehavior` = Bad). Note that in the second case, malicious peers must be colluding. 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`.

No collusion. We first focus on the non-colluding case where `BadPongBehavior` = Dead. Figures 16 and 17 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 combi-

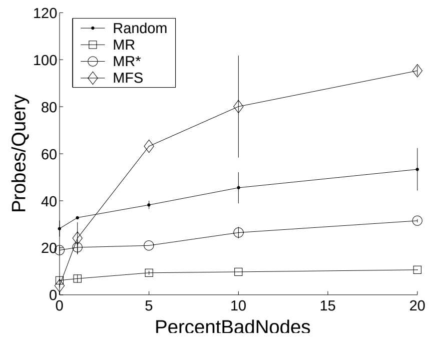

Figure 16: **Average probes per query increases as the number of malicious peers increases**

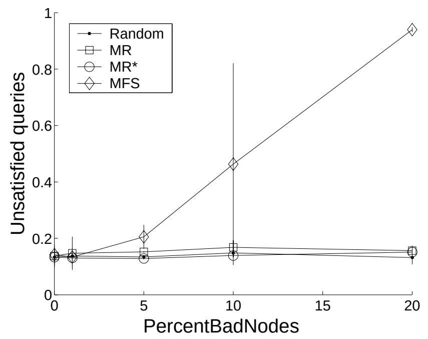

Figure 17: **Satisfaction decreases as the number of malicious peers increases**

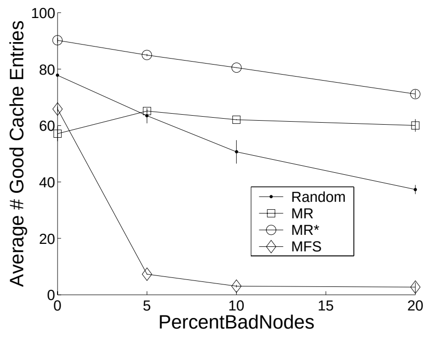

Figure 18: **Good peers' link caches become highly poisoned as `PercentBadPeers` increases for MFS only**

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

In Figure 18, we see the average number of good, or "unpoisoned", cache entries in a good peers' link cache, as `PercentBadPeers` increases. We see that for the MFS policy only, the number of good entries drops off dramatically as the number of malicious peers grows. Because good peers no longer have any good entries in their link caches, they can no longer receive many quality results for their queries.

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 when `BadPongBehavior` = Dead. As wee see in Figure 18, the average number of "unpoisoned" cache entries in a good peers' link cache is barefly affected by the number of malicious peers in the system.

The Random and MR* policies have good robustness as well. As expected, MR* has worse performance than MR, because MR* does not take advantage of other peers' experience. However, MR* outperforms Random because even a peer's limited, local experience with another peer is more useful than no information at all. Therefore, MR* will consistently outperform Random in terms of efficiency (number of probes), while maintaining the same effectiveness (satisfaction rate).

Because MR has the best performance by far for almost all values of `PercentBadPeers` in the non-collusion scenario, it is the preferred strategy. However, we will see next that if malicious peers collude, MR is no longer robust.

Collusion. If malicious peers collude, they can return each others' IP addresses in their Pongs, rather than dead IP addresses. Doing so is much more harmful to the overall system, because whereas dead IP addresses are always evicted after a single probe, IP addresses of malicious peers are not. Malicious peers simply appear as good peers who happen to return no query results. In addition, when probed, malicious peers will return the IP addresses of other malicious peers, which will then be inserted into the good peer's link cache. In contrast, because dead peers do not respond, they can not further "poison" the good peer's cache.

In Figures 19 and 20, we see the robustness of the different policies as `PercentBadPeers` is varied. We set

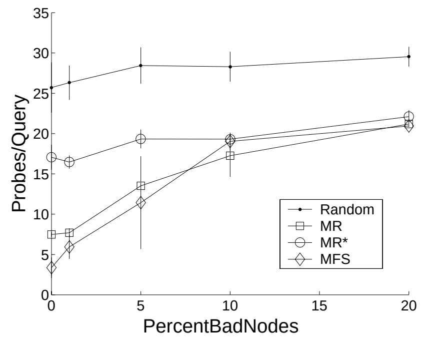

Figure 19: Average probes per query increases as the number of malicious peers increases

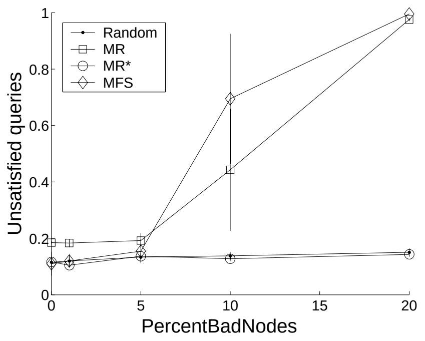

Figure 20: Satisfaction decreases as the number of malicious peers increases

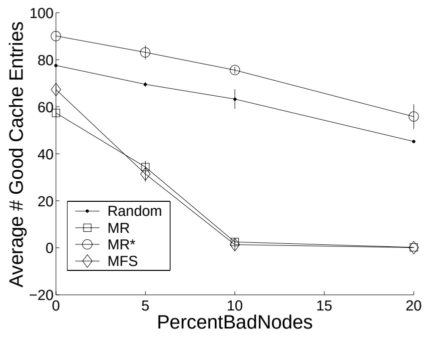

Figure 21: Good peers' link caches become highly poisoned as `PercentBadPeers` increases for MR and MFS

`BadPongBehavior` = Bad to model the case where malicious peers collude and return the addresses of other malicious peers in their Pongs. In these figures, we see that again, the MFS policy has the best performance when there are no malicious peers in the network. However, MFS is also not robust; when 20% of the network is malicious (`PercentBadPeers` = 20), 0% of the queries are satisfied (see Figure 20).

Likewise, MR has very poor robustness as well. Unlike in the non-colluding case, where `BadPongBehavior` = Dead, in these figures we see that both the average number of probes, and the unsatisfaction rate, increase rapidly as `PercentBadPeers` increases. Like MFS, when 20% of the network is malicious, MR yields 0% satisfied queries. This result may be surprising at first, because with MR, malicious peers are quickly evicted from good peers' link caches. Since malicious peers return no results, after one probe, the `NumRes` field for a malicious peer's is set to 0, making it a target for cache replacement. However, because probing a malicious peer once brings `PongSize` new malicious peers into the link cache, malicious peers enter the cache at a higher rate than at which they are evicted. Hence, even with the MR policy, good peers' caches remain highly "poisoned". We can corroborate this theory with Figure 21, which shows the average number of "unpoisoned" cache entries in a good peers' link cache as `PercentBadPeers` increases. As shown in this figure, good peers' link caches become highly poisoned (i.e., the number of good entries is small) as `PercentBadPeers` increases, for both the MR and MFS policies.

The Random and MR* policies once again exhibit good robustness, because they do not rely on information provided by other peers (number of files, number of results) to rank their cache entries. Furthermore, as before, MR* outperforms Random significantly, because it takes advantage of trusted, secure knowledge on the past performance of a given peer. However, although MR* is more robust than MR or MFS, when there are few malicious peers in the system, the performance of MR* is poor compared to that of MR and MFS. For example, at `PercentBadPeers` = 0, MR* requires 17 probes on average per query, while MFS requires just 4 probes, and MR requires 7.

In summary, the MR* policy is the best overall choice when malicious peers are a possibility. MR* outperforms Random consistently, but is much more robust than MR or MFS. Ideally, however, peers can learn to switch between MR and MR* if malicious peers are present. When few malicious peers are present, MR should be the policy of choice. If many malicious peers are present, the peer should use MR*. Although proving that a peer is malicious may be difficult, detecting malicious peers can be accomplished using heuristics – for example, if a group of peers constantly include each other in pongs, or if a peer consistently returns many dead IP addresses in its Pong. Hence, *detection* of malicious peers, and *adapting* policies based on this information, remains an interesting and important area of future work. The related issue of cache-poisoning prevention (without actually detecting bad peers) is addressed in [9].

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] E. Damiani, D. Vimercati, S. Paraboschi, P. Samarati, and F Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *In ACM CCS 2002*, 2003.

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

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

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

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

[12] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *In Proc. WWW*, 2003.

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

[14] LimeWire website. http://www.limewire.com.

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

[16] 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.

[17] 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*.

[18] 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.

[19] 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.

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

[21] 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.

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

[23] B. Yang and H. Garcia-Molina. Ppay: Micropayments for peer-to-peer systems. Technical report, Stanford University, 2003.