Making Gnutella-like P2P Systems Scalable

Yatin Chawathe
AT&T Labs–Research
yatin@research.att.com

Sylvia Ratnasamy
Intel Research
sylvia@intel-research.net

Lee Breslau
AT&T Labs–Research
breslau@research.att.com

Nick Lanham
UC Berkeley
nickl@cs.berkeley.edu

Scott Shenker[*]
ICSI
shenker@icsi.berkeley.edu

ABSTRACT

Napster pioneered the idea of peer-to-peer file sharing, and supported it with a centralized file search facility. Subsequent P2P systems like Gnutella adopted decentralized search algorithms. However, Gnutella's notoriously poor scaling led some to propose distributed hash table solutions to the wide-area file search problem. Contrary to that trend, we advocate retaining Gnutella's simplicity while proposing new mechanisms that greatly improve its scalability. Building upon prior research [1, 12, 21], we propose several modifications to Gnutella's design that dynamically adapt the overlay topology and the search algorithms in order to accommodate the natural heterogeneity present in most peer-to-peer systems. We test our design through simulations and the results show three to five orders of magnitude improvement in total system capacity. We also report on a prototype implementation and its deployment on a testbed.

Categories and Subject Descriptors

C.2 [**Computer Communication Networks**]: Distributed Systems

General Terms

Algorithms, Design, Performance, Experimentation

Keywords

Peer-to-peer, distributed hash tables, Gnutella

1. INTRODUCTION

The peer-to-peer file-sharing revolution started with the introduction of Napster in 1999. Napster was the first system to recognize that requests for popular content need not be sent to a central server but instead could be handled by the many hosts, or peers, that already possess the content. Such serverless peer-to-peer systems can achieve astounding aggregate download capacities without requiring any additional expenditure for bandwidth or server farms.[1] Moreover, such P2P file-sharing systems are *self-scaling* in that as more peers join the system to look for files, they add to the aggregate download capability as well.[2]

However, to make use of this self-scaling behavior, a node looking for files must find the peers that have the desired content. Napster used a centralized search facility based on file lists provided by each peer. By centralizing search (which does not require much bandwidth) while distributing download (which does), Napster achieved a highly functional hybrid design.

The resulting system was widely acknowledged as "the fastest growing Internet application ever"[4]. But RIAA's lawsuit forced Napster to shut down, and its various centralized-search successors have faced similar legal challenges. These centralized systems have been replaced by new *decentralized* systems such as Gnutella [8] that distribute both the download and search capabilities. These systems establish an overlay network of peers. Queries are not sent to a central site, but are instead distributed among the peers. Gnutella, the first of such systems, uses an *unstructured* overlay network in that the topology of the overlay network and placement of files within it is largely unconstrained. It floods each query across this overlay with a limited scope. Upon receiving a query, each peer sends a list of all content matching the query to the originating node. Because the load on each node grows linearly with the total number of queries, which in turn grows with system size, this approach is clearly not scalable.

Following Gnutella's lead, several other decentralized file-sharing systems such as KaZaA [23] have become popular. KaZaA is based on the proprietary Fasttrack technology which uses specially designated *supernodes* that have higher bandwidth connectivity. Pointers to each peer's data are stored on an associated supernode, and all queries are routed to supernodes. While this approach appears to offer better scaling than Gnutella, its design has been neither documented nor analyzed. Recently, there have been proposals to incorporate this approach into the Gnutella network [7]. Although some Gnutella clients now implement the supernode proposal, its scalability has neither been measured nor been analyzed.

That said, we believe that the supernode approach popularized by KaZaA is a step in the right direction for building scalable file-sharing systems. In this paper, we leverage this idea of exploiting node heterogeneity, but make the selection of "supernodes" and construction of the topology around them more dynamic and adap-

[*]Supported in part by NSF grants ITR-0205519, ANI-0207399, ITR-0121555, ITR-0081698, ITR-0225660 and ANI-0196514.

[1]For instance, 100,000 peers all connected at 56kbps can provide more aggregate download capacity than a single server farm connected by two OC-48 links.

[2]This self-scaling property is mitigated to some extent by the *free rider* problem observed in such systems [2].

tive. We present a new P2P file-sharing system, called Gia.[3] Like Gnutella and KaZaA, Gia is decentralized and unstructured. However, its unique design achieves an aggregate system capacity that is three to five orders of magnitude better than that of Gnutella as well as that of other attempts to improve Gnutella [12, 23]. As such, it retains the simplicity of an unstructured system while offering vastly improved scalability.

The design of Gia builds on a substantial body of previous work. As in the recent work by Lv *et al.* [12], Gia replaces Gnutella's flooding with random walks. Following the work of Adamic *et al.* [1], Gia recognizes the implications of the overlay network's topology while using random walks and therefore includes a topology adaptation algorithm. Similarly, the lack of flow control has been recognized as a weakness in the original Gnutella design [15], and Gia introduces a token-based flow control algorithm. Finally, like KaZaA, Gia recognizes that there is significant heterogeneity in peer bandwidth and incorporates heterogeneity into each aspect of our design.

While Gia does build on these previous contributions, Gia is, to our knowledge, the first open design that (a) combines all these elements, and (b) recognizes the fact that peers have capacity constraints and adapts its protocols to account for these constraints. Our simulations suggest that this results in a tremendous boost for Gia's system performance. Moreover, this performance improvement comes not just from a single design decision but from the synergy among the various design features.

We discuss Gia's design in Section 3, its performance in Section 4, and a prototype implementation and associated practical issues in Section 5. However, before embarking on the description of Gia, we first ask why not just use *Distributed Hash Tables* (DHTs).

2. WHY NOT DHTS?

Distributed Hash Tables are a class of recently-developed systems that provide hash-table-like semantics at Internet scale [24, 17, 26]. Much (although not all) of the original rationale for DHTs was to provide a scalable replacement for unscalable Gnutella-like file sharing systems. The past few years has seen a veritable frenzy of research activity in this field, with many design proposals and suggested applications. All of these proposals use *structured* overlay networks where both the data placement and overlay topology are tightly controlled. The hash-table-like lookup() operation provided by DHTs typically requires only $O(\log n)$ steps, whereas in comparison, Gnutella requires $O(n)$ steps to reliably locate a specific file.

Given this level of performance gain afforded by DHTs, it is natural to ask why bother with Gia when DHTs are available. To answer this question, we review three relevant aspects of P2P file sharing.

#1: P2P clients are extremely transient. Measured activity in Gnutella and Napster indicates that the median up-time for a node is 60 minutes [21].[4] For large systems of, say, 100,000 nodes, this implies a churn rate of over 1600 nodes coming and going per minute. Churn causes little problem for Gnutella and other systems that employ unstructured overlay networks as long as a peer doesn't become disconnected by the loss of all of its neighbors, and even in that case the peer can merely repeat the bootstrap procedure to re-join the network. In contrast, churn does cause significant overhead for DHTs. In order to preserve the efficiency and correctness of routing, most DHTs require $O(\log n)$ repair operations after each failure. *Graceless* failures, where a node fails without beforehand informing its neighbors and transferring the relevant state, require more time and work in DHTs to (a) discover the failure and (b) re-replicate the lost data or pointers. If the churn rate is too high, the overhead caused by these repair operations can become substantial and could easily overwhelm nodes with low-bandwidth dial-up connections.

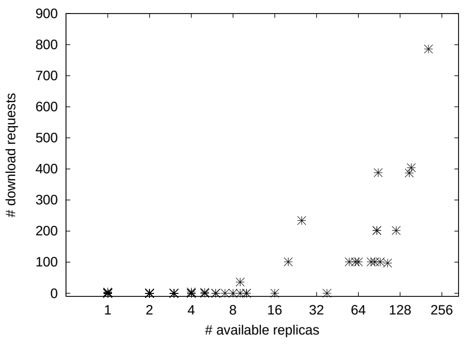

Figure 1: ***Most download requests are for well-replicated files.***

#2: Keyword searches are more prevalent, and more important, than exact-match queries. DHTs excel at supporting exact-match lookups: given the exact name of a file, they translate the name into a key and perform the corresponding lookup(key) operation. However, DHTs have been less successful in supporting keyword searches: given a sequence of keywords, find files that match them. The current use of P2P file-sharing systems, which revolves around sharing music and video, requires such keyword matching. For example, to find a remix version of Madonna's song "Ray of Light," a user typically submits a search of the form "madonna ray of light remix" and expects the file-sharing system to locate files that match all of the keywords in the search query. This is especially important since there is no unambiguous naming convention for file names in P2P systems, and thus often the same piece of content is stored by different nodes under several (slightly different) names. Supporting such keyword searching on top of DHTs is a non-trivial task. For example, the typical approach [11, 14, 18, 25] of constructing an inverted index per keyword can be expensive to maintain in the face of frequent node (and hence file) churn. This is only further complicated by the additional caching algorithms needed to avoid overloading nodes that store the index for popular keywords. In contrast, Gnutella and other similar systems effortlessly support keyword searches since all such searches are executed locally on a node-by-node basis.

#3: Most queries are for hay, not needles. DHTs have *exact* recall, in that knowing the name of a file allows you to find it, even if there is only a single copy of that file in the system. In contrast, Gnutella cannot reliably find single copies of files unless the flooded query reaches all nodes; we call such files *needles*. However, we expect that most queries in the popular P2P file-sharing systems are for relatively well-replicated files, which we call *hay*. By the very nature of P2P file-sharing, if a file is requested frequently, then as more and more requesters download the file to their machines, there will be many copies of it within the system. We call such systems, where most queries are for well-replicated content, *mass-market* file-sharing systems.

Gnutella can easily find well-replicated files. Thus, if most searches are for hay, not needles, then Gnutella's lack of exact recall is not a significant disadvantage. To verify our conjecture that most queries

[3]Gia is short for gianduia, which is the generic name for the hazelnut spread, Nutella.
[4]We understand that there is some recently published work [3] that questions the exact numbers in this study, but the basic point remains that the peer population is still quite transient.

are indeed for hay, we gathered traces of queries and download requests using an instrumented Gnutella client. Our tracing tool crawled the Gnutella network searching for files that match the top 50 query requests seen. After gathering the file names and the number of available copies of each of these files, the tool turned around and offered the same files for download to other Gnutella clients. We then measured the number of download requests seen by the tracing tool for this offered content. Figure 1 shows the distribution of the download requests versus the number of available replicas. We notice that most of the requests correspond to files that have a large number of available replicas.[5] For example, half of the requests were for files with more than 100 replicas, and approximately 80% of the requests were for files with more than 80 replicas.

In summary, Gnutella-like designs are more robust in the face of transients and support general search facilities, both important properties to P2P file sharing. They are less adept than DHTs at finding needles, but this may not matter since most P2P queries are for hay. Thus, we conjecture that for mass-market file-sharing applications, improving the scalability of unstructured P2P systems, rather than turning to DHT-based systems, may be the better approach.

3. GIA DESIGN

Gnutella-like systems have one basic problem: when faced with a high aggregate query rate, nodes quickly become overloaded and the system ceases to function satisfactorily. Moreover, this problem gets worse as the size of the system increases. Our first goal in designing Gia is to create a Gnutella-like P2P system that can handle much higher aggregate query rates. Our second goal is to have Gia continue to function well at arbitrary system sizes. To achieve this scalability, Gia strives to avoid overloading any of the nodes by explicitly accounting for their capacity constraints. In an earlier workshop paper [13], we presented a preliminary proposal for incorporating capacity awareness into Gnutella. In our current work, we refine those ideas and present a thorough design, detailed algorithms, and a prototype implementation of the new system. We begin with an overview of the reasoning behind our system design and then provide a detailed discussion of the various components and protocols.

3.1 Design Rationale

The Gnutella protocol [6] uses a flooding-based search method to find files within its P2P network. To locate a file, a node queries each of its neighbors, which in turn propagate the query to their neighbors, and so on until the query reaches all of the clients within a certain radius from the original querier. Although this approach can locate files even if they are replicated at an extremely small number of nodes, it has obvious scaling problems. To address this issue, Lv *et al.* [12] proposed replacing flooding with random walks. Random walks are a well-known technique in which a query message is forwarded to a randomly chosen neighbor at each step until sufficient responses to the query are found. Although they make better utilization of the P2P network than flooding, they have two associated problems:

1. A random walk is essentially a blind search in that at each step a query is forwarded to a random node without taking into account any indication of how likely it is that the node will have responses for the query.

2. If a random walker query arrives at a node that is already overloaded with traffic, it may get queued for a long time before it is handled.

Adamic *et al.* [1] addressed the first problem by recommending that instead of using purely random walks, the search protocol should bias its walks toward high-degree nodes. The intuition behind this is that if we arrange for neighbors to be aware of each other's shared files, high-degree nodes will have (pointers to) a large number of files and hence will be more likely to have an answer that matches the query. However, this approach ignores the problem of overloaded nodes. In fact, by always biasing the random walk towards high-degree nodes, it can exacerbate the problem if the high-degree node does not have the capacity to handle a large number of queries.

The design of Gia, on the other hand, explicitly takes into account the capacity constraints associated with each node in the P2P network. The capacity of a node depends upon a number of factors including its processing power, disk latencies, and access bandwidth. It is well-documented that nodes in networks like Gnutella exhibit significant heterogeneity in terms of their capacity to handle queries [21]. Yet, none of the prior work on scaling Gnutella-like systems leverages this heterogeneity. In the design of Gia, we explicitly accommodate (and even exploit) heterogeneity to achieve better scaling. The four key components of our design are summarized below:

- A *dynamic topology adaptation* protocol that puts most nodes within short reach of high capacity nodes. The adaptation protocol ensures that the well-connected (*i.e.,* high-degree) nodes, which receive a large proportion of the queries, actually have the capacity to handle those queries.

- An *active flow control* scheme to avoid overloaded hot-spots. The flow control protocol explicitly acknowledges the existence of heterogeneity and adapts to it by assigning flow-control tokens to nodes based on available capacity.

- *One-hop replication* of pointers to content. All nodes maintain pointers to the content offered by their immediate neighbors. Since the topology adaptation algorithm ensures a congruence between high capacity nodes and high degree nodes, the one-hop replication guarantees that high capacity nodes are capable of providing answers to a greater number of queries.

- A *search protocol* based on biased random walks that directs queries towards high-capacity nodes, which are typically best able to answer the queries.

3.2 Detailed Design

The framework for the Gia client and protocols is modeled after the current Gnutella protocol [6]. Clients connect to each other using a three-way handshake protocol. All messages exchanged by clients are tagged at their origin with a *globally unique identifier* or GUID, which is a randomly generated sequence of 16 bytes. The GUID is used to track the progress of a message through the Gia network and to route responses back to the originating client.

We extend the Gnutella protocol to take into account client capacity and network heterogeneity. For this discussion, we assume that client capacity is a quantity that represents the number of queries that the client can handle per second. In practice, the capacity will have to be determined as a function of a client's access bandwidth, processing power, disk speed, etc. We discuss the four protocol components in detail below.

[5]Note that since the tracing tool only captures the download requests that came directly to it, we miss all of the requests that went to the other nodes that also had copies of the same file. Thus our numbers can only be a lower bound on how popular well-replicated content is.

```
Let $C_i$ represent capacity of node $i$
if $num\_nbrs_X + 1 \leq max\_nbrs$ then {we have room}
    ACCEPT $Y$; return

{we need to drop a neighbor}
$subset \leftarrow i \ \forall\, i \in nbrs_X$ such that $C_i \leq C_Y$
if no such neighbors exist then
    REJECT $Y$; return
candidate $Z \leftarrow$highest-degree neighbor from $subset$

if $(C_Y > max(C_i \,\forall\, i \in nbrs_X)\,)$ {$Y$ has higher capacity}
or $(num\_nbrs_Z > num\_nbrs_Y + H)$ {$Y$ has fewer nbrs}
 then
    DROP $Z$; ACCEPT $Y$
else
    REJECT $Y$
```

Algorithm 1: $pick\_neighbor\_to\_drop(X, Y)$:
When node X tries to add Y as a new neighbor, determine whether there is room for Y. If not, pick one of X's existing neighbors to drop and replace it with Y. (In the algorithm, H represents a hysteresis factor.)

3.2.1 Topology Adaptation

The topology adaptation algorithm is the core component that connects the Gia client to the rest of the network. In this section, we provide an overview of the adaptation process, while leaving the details of some of the specific mechanisms for discussion later in Section 5. When a node starts up, it uses bootstrapping mechanisms similar to those in Gnutella to locate other Gia nodes. Each Gia client maintains a *host cache* consisting of a list of other Gia nodes (their IP address, port number, and capacity). The host cache is populated throughout the lifetime of the client using a variety of rendezvous mechanisms including contacting well-known web-based host caches [5] and exchanging host information with neighbors through PING-PONG messages [6]. Entries in the host cache are marked as dead if connections to those hosts fail. Dead entries are periodically aged out.

The goal of the topology adaptation algorithm is to ensure that high capacity nodes are indeed the ones with high degree and that low capacity nodes are within short reach of higher capacity ones. To achieve this goal, each node independently computes a *level of satisfaction* (S). This is a quantity between 0 and 1 that represents how satisfied a node is with its current set of neighbors. A value of $S = 0$ means that the node is quite dissatisfied, while $S = 1$ suggests that the node is fully satisfied. As long as a node is not fully satisfied, the topology adaptation continues to search for appropriate neighbors to improve the satisfaction level. Thus, when a node starts up and has fewer than some pre-configured minimum number of neighbors, it is in a dissatisfied state ($S = 0$). As it gathers more neighbors, its satisfaction level rises, until it decides that its current set of neighbors is sufficient to satisfy its capacity, at which point the topology adaptation becomes quiescent. In Section 5.2, we describe the details of the algorithm used to compute the satisfaction level.

To add a new neighbor, a node (say X) randomly selects a small number of candidate entries from those in its host cache that are not marked dead and are not already neighbors. From these randomly chosen entries, X selects the node with maximum capacity greater than its own capacity. If no such candidate entry exists, it selects one at random. Node X then initiates a three-way handshake to the selected neighbor, say Y.

During the handshake, each node makes a decision whether or not to accept the other node as a new neighbor based upon the capacities and degrees of its existing neighbors and the new node. In order to accept the new node, we may need to drop an existing neighbor. Algorithm 1 shows the steps involved in making this determination. The algorithm works as follows. If, upon accepting the new connection, the total number of neighbors would still be within a pre-configured bound max_nbrs, then the connection is automatically accepted. Otherwise, the node must see if it can find an appropriate existing neighbor to drop and replace with the new connection.

X always favors Y and drops an existing neighbor if Y has higher capacity than all of X's current neighbors. Otherwise, it decides whether to retain Y or not as follows. From all of X's neighbors that have capacity less than or equal to that of Y, we choose the neighbor Z that has the highest degree. This neighbor has the least to lose if X drops it in favor of Y. The neighbor will be dropped *only* if the new node Y has fewer neighbors than Z. This ensures that we do not drop already poorly-connected neighbors (which could get disconnected) in favor of well-connected ones.[6] The topology adaptation algorithm thus tries to ensure that the adaptation process makes forward progress toward a stable state. Results from experiments measuring the topology adaptation process are discussed later in Section 5.4.

3.2.2 Flow control

To avoid creating hot-spots or overloading any one node, Gia uses an active flow control scheme in which a sender is allowed to direct queries to a neighbor only if that neighbor has notified the sender that it is willing to accept queries from the sender. This is in contrast to most proposed Gnutella flow-control mechanisms [15], which are reactive in nature: receivers drop packets when they start to become overloaded; senders can infer the likelihood that a neighbor will drop packets based on responses that they receive from the neighbor, but there is no explicit feedback mechanism. These technique may be acceptable when queries are flooded across the network, because even if a node drops a query, other copies of the query will propagate through the network. However, Gia uses random walks (to address scaling problems with flooding) to forward a single copy of each query. Hence, arbitrarily dropping queries is not an appropriate solution.

To provide better flow control, each Gia client periodically assigns flow-control tokens to its neighbors. Each token represents a single query that the node is willing to accept. Thus, a node can send a query to a neighbor only if it has received a token from that neighbor, thus avoiding overloaded neighbors. In the aggregate, a node allocates tokens at the rate at which it can process queries. If it receives queries faster than it can forward them (either because it is overloaded or because it has not received enough tokens from its neighbors), then it starts to queue up the excess queries. If this queue gets too long, it tries to reduce the inflow of queries by lowering its token allocation rate.

To provide an incentive for high-capacity nodes to advertise their true capacity, Gia clients assign tokens in proportion to the neighbors' capacities, rather than distributing them evenly between all neighbors. Thus, a node that advertises high capacity to handle incoming queries is in turn assigned more tokens for its own outgoing queries. We use a token assignment algorithm based on Start-time Fair Queuing (SFQ) [9]. Each neighbor is assigned a fair-queuing weight equal to its capacity. Neighbors that are not using any of their assigned tokens are marked as inactive and the left-over capacity is automatically redistributed proportionally between the remaining neighbors. As neighbors join and leave, the SFQ algorithm recon-

[6]To avoid having X flip back and forth between Y and Z, we add a level of hysteresis: we drop Z and add Y only if Y has at least H fewer neighbors than Z, where H represents the level of hysteresis. In our simulations and implementation, we set the value of H to 5.

figures its token allocation accordingly.[7] Token assignment notifications can be sent to neighbors either as separate control messages or by piggy-backing on other messages.

3.2.3 One-hop Replication

To improve the efficiency of the search process, each Gia node actively maintains an index of the content of each of its neighbors. These indices are exchanged when neighbors establish connections to each other, and periodically updated with any incremental changes. Thus, when a node receives a query, it can respond not only with matches from its own content, but also provide matches from the content offered by all of its neighbors. When a neighbor is lost, either because it leaves the system, or due to topology adaptation, the index information for that neighbor gets flushed. This ensures that all index information remains mostly up-to-date and consistent throughout the lifetime of the node.

3.2.4 Search Protocol

The combination of topology adaptation (whereby high capacity nodes have more neighbors) and one-hop replication (whereby nodes keep an index of their neighbors' shared files) ensures that high capacity nodes can typically provide useful responses for a large number of queries. Hence, the Gia search protocol uses a *biased* random walk: rather than forwarding incoming queries to randomly chosen neighbors, a Gia node selects the highest capacity neighbor for which it has flow-control tokens and sends the query to that neighbor. If it has no tokens from any neighbors, it queues the query until new tokens arrive.

We use TTLs to bound the duration of the biased random walks and book-keeping techniques to avoid redundant paths. With book-keeping, each query is assigned a unique GUID by its originator node. A node remembers the neighbors to which it has already forwarded queries for a given GUID. If a query with the same GUID arrives back at the node, it is forwarded to a different neighbor. This reduces the likelihood that a query traverses the same path twice. To ensure forward progress, if a node has already sent the query to all of its neighbors, it flushes the book-keeping state and starts re-using neighbors.

Each query has a MAX_RESPONSES parameter, the maximum number of matching answers that the query should search for. In addition to the TTL, query duration is bounded by MAX_RESPONSES. Every time a node finds a matching response for a query, it decrements the MAX_RESPONSES in the query. Once MAX_RESPONSES hits zero, the query is discarded. Query responses are forwarded back to the originator along the reverse-path associated with the query. If the reverse-path is lost due to topology adaptation or if queries or responses are dropped because of node failure, we rely on recovery mechanisms described later in Section 5.3 to handle the loss.

Finally, since a node can generate a response either for its own files or for the files of one of its neighbors, we append to the forwarded query the addresses of the nodes that own those files. This ensures that the query does not produce multiple redundant responses for the same instance of a file; a response is generated only if the node that owns the matching file is not already listed in the query message.

4. SIMULATIONS

In this section, we use simulations to evaluate Gia and compare its performance to two other unstructured P2P systems. Thus our simulations refer to the following four models:

[7]Details of the SFQ algorithm for proportional allocation can be found in [9].

Capacity level	Percentage of nodes
1x	20%
10x	45%
100x	30%
1000x	4.9%
10000x	0.1%

Table 1: Gnutella-like node capacity distributions.

- FLOOD: Search using TTL-scoped flooding over random topologies. This represents the Gnutella model.
- RWRT: Search using random walks over random topologies. This represents the recommended search technique suggested by Lv *et al.* [12] for avoiding the scalability problems with flooding.
- SUPER: Search using supernode mechanisms [7, 23]. In this approach, we classify nodes as supernodes and non-supernodes. Queries are flooded only between supernodes.
- GIA: Search using the Gia protocol suite including topology adaptation, active flow control, one-hop replication, and biased random walks.

We first describe our simulation model and the metrics used for evaluating the performance of our algorithms. Then we report the results from a range of simulations. Our experiments focus on the aggregate system behavior in terms of its capacity to handle queries under a variety of conditions. We show how the individual components of our system (topology adaptation, flow control, one-hop replication, and searches based on biased random walks) and the synergies between them affect the total system capacity. Due to space limitations, we do not present detailed results evaluating tradeoffs within each design component.

4.1 System Model

To capture the effect of query load on the system, the Gia simulator imposes capacity constraints on each of the nodes within the system. We model each node i as possessing a capacity C_i, which represents the number of messages (such as queries and add/drop requests for topology adaptation) that it can process per unit time. If a node receives queries from its neighbors at a rate higher than its capacity C_i (as can happen in the absence of flow control), then the excess queries are modeled as being queued in connection buffers until the receiving node can read the queries from those buffers.

For most of our simulations, we assign capacities to nodes based on a distribution that is derived from the measured bandwidth distributions for Gnutella as reported by Saroiu *et al.* [21]. Our capacity distribution has five levels of capacity, each separated by an order of magnitude as shown in Table 1. As described in [21], this distribution reflects the reality that a fair fraction of Gnutella clients have dial-up connections to the Internet, the majority are connected via cable-modem or DSL and a small number of participants have high speed connections. For the SUPER experiments, nodes with capacities 1000x and 10000x are designated as supernodes.

In addition to its capacity, each node i is assigned a query generation rate q_i, which is the number of queries that node i generates per unit time. For our experiments, we assume that all nodes generate queries at the same rate (bounded, of course, by their capacities). When queries need to be buffered, they are held in queues. We model all incoming and outgoing queues as having infinite length. We realize that, in practice, queues are not infinite, but we make this assumption since the effect of dropping a query and adding it to an arbitrarily long queue is essentially the same.

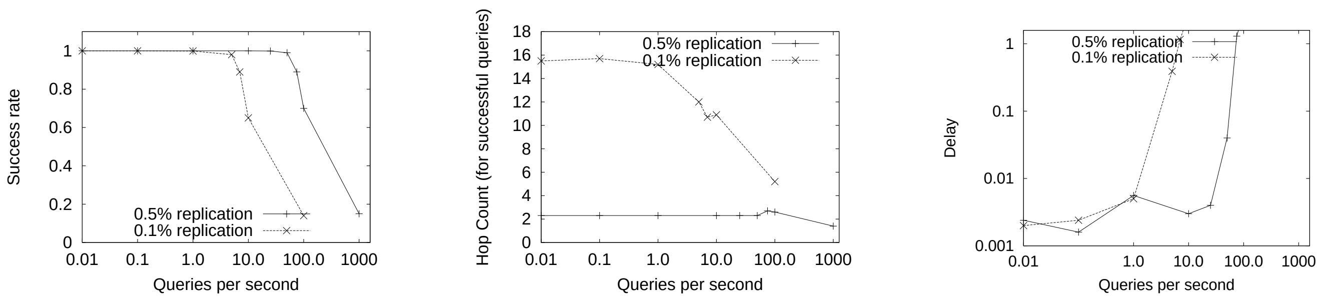

Figure 2: Success rate, hop-count and delay under increasing query load for a 10,000 node Gia network.

Queries are modeled as searching for specific keywords. Each keyword maps on to a set of files. Files are randomly replicated on nodes. All files associated with a specific keyword are potential *answers* for a query with that keyword. We use the term *replication factor* to refer to the fraction of nodes at which answers to queries reside. Thus, performing a query for a keyword that has a replication factor of 1% implies that an answer to this query can be found at 1% of the nodes in the system. In a deployed system, real search traffic will include many different queries covering a range of replication factors simultaneously. However, each search process proceeds largely independently (aside from delays within queues and the actions of flow control). Hence, rather than having to pick a specific distribution of queries, each looking for keywords with their own replication factors, we focus on a stream of queries all with a particular replication factor and study how our results vary as we change the replication factor.

We begin our simulations with a randomly connected topology.[8] The GIA simulations use topology adaptation to reconfigure this initial topology. The algorithms use two pre-configured parameters: min_nbrs and max_nbrs. For all of our experiments, we use $min\_nbrs = 3$. We set max_nbrs to 128. However, there is an additional constraint on max_nbrs. To avoid mid- or low-capacity nodes from gathering so many neighbors that their capacity is too finely divided, we require that $\frac{C}{num\_nbrs} \geq$ some min_alloc, where min_alloc represents the finest level of granularity into which we are willing to split a node's capacity. With this additional constraint, we note that for each node, $max\_nbrs = \min(max\_nbrs, \lfloor \frac{C}{min\_alloc} \rfloor)$. After some preliminary simulations that tested the performance of the GIA topology adaptation for different values of min_alloc, we settled on $min\_alloc = 4$. All control traffic generated by the topology adaptation and other components is modeled as consuming resources: one unit of capacity per message. Thus, the simulator indirectly captures the impact of control traffic on the overall performance of the system.

For RWRT and FLOOD, there is no topology adaptation; we use a random graph. We know that Gnutella networks, in fact, exhibit properties similar to power-law graphs [19]. However, there is no assurance that high-degree nodes in the skewed Gnutella distribution are also high-capacity nodes. In fact, in the absence of an explicit congruence of high capacity with high degree, a random walk will cause the high-degree nodes to get overloaded. Comparing a random walk on such a topology to GIA would unfairly bias the results against the random walk. Hence, for RWRT, we choose to use a purely random topology with uniform degree distributions, which mitigates this problem. The RWRT performance on such a uniformly random graph is independent of the degree of the individual nodes; all nodes will be visited with the same probability. On the other hand, the performance of FLOOD does depend on degree and in fact worsens with higher degree. For our experiments, we thus chose uniformly random graphs with an average degree of eight. This choice is ad hoc, but refects a decision to avoid unnecessarily biasing against RWRT and FLOOD.

[8]For the SUPER experiments, we use a topology where supernodes set up random connections among themselves. In addition, all non-supernodes connect to one supernode at random.

On average, the diameter of our random graphs is 7. Thus, for FLOOD and SUPER, we set the TTL for queries to 10 to ensure that queries do not get artificially limited. For RWRT and GIA, the TTL is set to a larger value (1024), but in this case setting the right TTL value is not as crucial because the random walks terminate when they find the required number of responses.

Although the simulator models the behavior of the various protocols discussed in Section 3, it does not capture individual packet-level behavior nor does it account for any of the vagaries in network behavior caused by background traffic. We do this because our point is not to quantify the *absolute* performance of the algorithm in real-world terms, but to evaluate the *relative* performance of the various design choices. In Section 5.4, we present some preliminary results that report on our experiences with implementing and deploying Gia in the wide-area Internet.

4.2 Performance Metrics

To measure the effect of load on the system, we looked at three aspects of the system's performance as a function of the offered load: the *success rate* measured as the fraction of queries issued that successfully locate the desired files[9], the *hop-count* measured as the number of hops required to locate the requested files, and the *delay* measured as the time taken by a query from start to finish. Figure 2 shows the success rate, hop-count and delay under increasing query load for a 10,000 node network running the Gia system. For these graphs, as in the remainder of our simulations, when we mention a query load of say 0.1, we mean that *every* node in the system issues 0.1 queries per unit time (bounded by the node's capacity, of course). As each of the graphs in the figure shows, when the query load increases, we notice a sharp "knee" in the curves beyond which the success rate drops sharply and delays increase rapidly. The hop-count holds steady until the knee-point and then decreases. The reason for this decrease is that hop-count is measured only for successful queries; under increasing load, successful queries tend to be those where the requested file is located within a few hops from the originator of the query. These graphs depict the existence of a knee in the GIA model; our simulations with RWRT, FLOOD, and SUPER over a range of replication factors revealed the same kind of behavior, although at different query loads.

Ideally, we want a system that achieves a high success rate while

[9]A query is deemed unsuccessful if at the end of the simulation it has generated no responses and is stuck in queues within overloaded nodes.

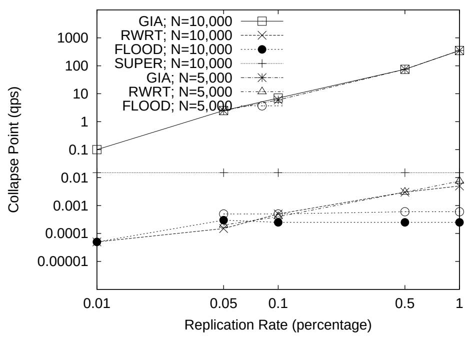

Figure 3: Comparison of collapse point for the different algorithms at varying replication rates and different system sizes.

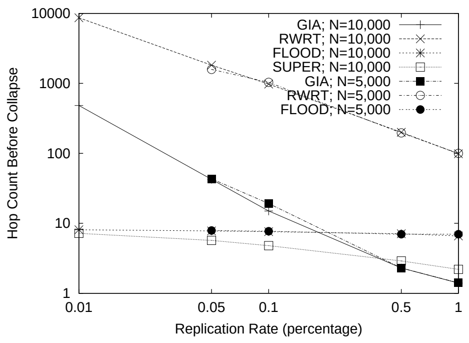

Figure 4: Hop-count before collapse.

maintaining a low hop-count and delay. To do so, the system must operate before the knee shown in the graphs above. Consequently, we define the following metrics for use in our evaluation:

Collapse Point (CP): the per node query rate at the knee, which we define as the point beyond which the success rate drops below 90%. This metric reflects total system capacity.

Hop-count before collapse (CP-HC): the average hop-count prior to collapse.

We do not retain delay as a metric since the effect of increasing delay is effectively captured by the collapse point.

4.3 Performance Comparison

We compare the behavior of GIA, RWRT, SUPER and FLOOD under varying replication factors and different system sizes up to 10,000 nodes. We measured the CP and CP-HC under increasing replication factors. In Figures 3 and 4, we plot the results for systems with 5,000 and 10,000 nodes. Experiments with additional system sizes yielded results consistent with those presented here; we omit them from the graphs for clarity. For a 10,000 node system we simulate down to 0.01% replication since that corresponds to a single matching answer in the entire system for any query. Likewise, for 5,000 nodes we simulate down to 0.05% replication. We believe that a replication factor of 0.01% where only one in 10,000 nodes holds the answer to a query represents a fairly pessimistic test scenario. Each query in these experiments runs until it finds one matching answer. This represents the case where the query originator sets the MAX_RESPONSES parameter (see Section 3.2.4) in the query to 1. In reality, most users expect a query to return multiple answers; we will look at that scenario later. For GIA and RWRT, we measure the average hop-count of all of the queries. Since for SUPER and FLOOD a query gets replicated at each hop, it is hard to define a consistent hop-count for the entire query; hence, we measure the hop-count as the number of hops taken to find the first answer.

Recall that our first goal in designing Gia was to enable it to handle a much higher aggregate query rate than Gnutella. The most obvious, and important, observation from Figures 3 and 4 is that the aggregate system capacity (as defined by the collapse point) is 3 to 5 orders of magnitude higher than either FLOOD or RWRT. Even when compared to the supernode approach, Gia does better especially at higher replication rates. This is not surprising since the flooding techniques used within supernodes limit their scalability. Thus, our goal of improving system capacity with Gia is clearly achieved. Our second goal was that Gia retain this ability to handle high aggregate query rates for systems of arbitrary sizes. As can be observed in the graphs, this goal is also satisfied. GIA's (and RWRT's) scaling behavior is determined by the replication factor. That is, at a fixed replication factor, the CP and CP-HC are largely unaffected by system size. This is to be expected since the replication factor is the *percentage* of nodes at which answers are located. Thus, the performance figures we show here apply to *arbitrarily large* system sizes.

There are several other performance results of note.

- At higher replication factors, RWRT performs better than FLOOD by approximately two orders of magnitude but is comparable to FLOOD at lower replication rates. This follows from the fact that at low replication rates, to find a matching answer RWRT may have to visit all of the nodes in the system just like FLOOD.
- GIA achieves extremely low hop-counts at higher replication because, in such cases, high capacity nodes are quite likely to hold answers and these are quickly discovered by biased walks. However, at low replication some queries may have to travel far beyond the high capacity nodes resulting in higher hop-counts. FLOOD and SUPER achieve consistently low hop-counts (the number of hops to find the *first* matching answer), while the hop-count for RWRT is inversely proportional to the replication factor, since RWRT essentially amounts to random probing.
- The performance of FLOOD degrades with increasing system size. This is because, in FLOOD, each query is propagated to every other node in the system. With increasing number of nodes, there are more total number of queries in the system, and hence a greater query load arriving at each node. This causes the collapse point to fall as the system size increases. We observed a similar effect with SUPER[10].

These experiments clearly demonstrate GIA's scalability relative to RWRT, SUPER and FLOOD. However, these experiments are limited to queries where the search terminates after finding a single matching answer. In reality, most users expect a query to return multiple answers. We now look at (a) how our results generalize beyond this single case, and (b) how the different design components contribute to this enormous performance boost for GIA.

4.4 Multiple Search Responses

In this section we look at how the collapse points (and the associated hop-counts) change for our different system models based upon the desired number of responses for a query. Recall from Section 3.2.4 that a query includes a MAX_RESPONSES parameter that indicates how many responses should be sent back to the originator of the query before ending the query. The MAX_RESPONSES parameter is useful only in the context of GIA and RWRT. For FLOOD and SUPER, queries get flooded through the network, and so MAX_RESPONSES has no effect on their behavior.

[10]For clarity, those graphs are left out.

Algorithm	Collapse Point	Hop-count
GIA	7	15.0
GIA – OHR	0.004	8570
GIA – BIAS	6	24.0
GIA – TADAPT	0.2	133.7
GIA – FLWCTL	2	15.1

Algorithm	Collapse Point	Hop-count
RWRT	0.0005	978
RWRT + OHR	0.005	134
RWRT + BIAS	0.0015	997
RWRT + TADAPT	0.001	1129
RWRT + FLWCTL	0.0006	957

Table 4: Factor analysis for GIA and RWRT with 10,000 modes and 0.1% replication. We measure GIA with each of the following components removed, and RWRT with each of those components added: one-hop replication (OHR), biased random walks (BIAS), topology adaptation (TADAPT), and flow-control (FLWCTL)

Repl. factor	MAX_ RESP.	GIA CP (CP-HC)	RWRT CP (CP-HC)	FLOOD CP	SUPER CP
1%	1	350 (1.4)	0.005 (98.7)	0.00025	0.015
1%	10	8 (12.5)	0.0004 (1020)	0.00025	0.015
1%	20	2.5 (28)	0.00015 (2157)	0.00025	0.015

Table 2: CP decreases with increasing numbers of requested answers (MAX_RESPONSES). The corresponding hop-counts before collapse for each case are shown in parentheses. Since hop-counts are ambiguous for FLOOD and SUPER when there are multiple responses, we ignore CP-HC for those cases.

Repl. factor	MAX_ RESPONSES	GIA CP	RWRT CP	FLOOD CP	SUPER CP
1%	10	8	0.0004	0.00025	0.015
0.1%	1	7	0.0005	0.00025	0.015
1%	20	2.5	0.00015	0.00025	0.015
0.05%	1	2.5	0.00015	0.00025	0.015

Table 3: A search for k responses at r% replication is equivalent to one for a single answer at $\frac{r}{k}$% replication.

Table 2 shows the CP for all four system models for a 10,000 node system at a replication factor of 1%. For RWRT and GIA, higher values of MAX_RESPONSES imply that the query needs to search through the network longer before it ends. This results in a higher effective hop-count for each query and as a result causes each query to utilize more of the available system capacity. As shown by the CP values in the table, this effectively reduces the overall system capacity. As expected, varying MAX_RESPONSES has no effect on the SUPER and FLOOD models.

As seen earlier in Figure 3, the collapse point also depends on the replication factor. When files are replicated at fewer nodes, queries must on average visit more nodes to find them. As a result, the collapse point drops with decreasing replication factors. In fact, we find that the performance of a query for k MAX_RESPONSES at a replication factor of r is equivalent to that of a query for a single response at a correspondingly lower replication factor of $\frac{r}{k}$. This is depicted in Table 3. With all four system models, searching for 10 answers at a replication factor of 1.0% yields a CP almost identical to that obtained by searching for a single answer at a replication factor of 0.1%. Likewise, searching for 20 answers at 1% replication yields the same CP as searching for a single answer at 0.05% replication.

Given this result, we model the rest of our GIA and RWRT simulations for the sake of simplicity with searches that terminate after finding the first answer for their queries. This does not change the nature of our results but makes it simpler to analyze the system and is sufficient to bring out the significant differences between the various designs.

4.5 Factor Analysis

Our results in Section 4.3 indicate that GIA outperforms RWRT, SUPER and FLOOD by several orders of magnitude in terms of the query load that it can successfully sustain. We now turn our attention to looking at how the individual components of GIA (topology adaptation, flow control, one-hop replication, and biased random walks) influence this performance gain. Many researchers have proposed schemes for improving Gnutella's scalability that use one or more of the GIA components. What distinguishes GIA from most other schemes is the combination of all of the components into a comprehensive system design that, unlike previous work, adapts each component to be "capacity-sensitive".

In this section, we show that it is not any single component, but in fact, the *combination* of them all that provides GIA this large performance advantage. We show that each of GIA's design components is vital to its performance, and yet, the addition of any single GIA component to RWRT does not significantly close the performance gap between GIA and RWRT. We do not consider FLOOD since the primary design leap from FLOOD to GIA is in the transition from the use of floods to the use of random walks, the effect of which is already captured by the basic RWRT. Similarly, SUPER is just one step toward the GIA design that includes some amount of one-hop replication and an ad-hoc awareness of node heterogeneity. Here, we instead examine the performance of GIA upon removing each of its four design components one at a time and compare it to the behavior of RWRT if we were to add those design components to it one at a time.

Table 4 shows the result of this factor analysis for 10,000 nodes at a replication of 0.1%. At first glance, one may conclude that GIA gets most of its performance gain from the one-hop replication, since removing one-hop replication from GIA severely impacts its performance. However, adding one-hop replication to RWRT only improves the CP by a single order of magnitude while GIA as a whole offers a CP that is over four orders of magnitude greater than with RWRT. It is the combination of topology adaptation, biased-random walks and flow-control in addition to the one-hop replication that gives GIA its enormous performance gain over RWRT.

Biasing the random walk appears to be of little consequence to GIA's performance. This is because at high query loads (*i.e.,* close to CP), the flow-control component serves to divert load towards any available capacity (which is typically in the high capacity nodes), and thus functions akin to the biased walks. However, under lower query loads, when all nodes are lightly loaded, it is the biased walk that helps to direct queries rapidly to high capacity nodes.

4.6 Effect of Heterogeneity

Since GIA is explicitly designed to be sensitive to node capacities, we now examine the impact of heterogeneity on system perfor-

Algorithm	Collapse Point	Hop-count
GIA w/ Gnutella capacity distribution	7	15.0
GIA w/ uniform capacity distribution	2	46.0
RWRT w/ Gnutella capacity distribution	0.0005	978
RWRT w/ uniform capacity distribution	0.0525	987

Table 5: Impact of heterogeneity; 10,000 nodes, 0.1% replication

mance. Table 5 compares the performance of GIA and RWRT with node capacities drawn from the Gnutella-like capacity distribution to the case where all nodes have identical capacities equal to the average node capacity from the Gnutella distribution. The CP in GIA improves when nodes have heterogeneous capacities. In contrast, we see that RWRT is not tolerant of heterogeneity and the CP drops by over two orders of magnitude relative to the uniform capacity case. While the CP-HC remains the same for RWRT in both cases (as one would expect), the hop-count for GIA drops since the biased random walks start directing queries towards the high-capacity nodes.

4.7 Robustness

Our results so far have shown that Gia performs significantly better than previous unstructured P2P file sharing systems. In this section, we show that Gia can sustain this performance in the face of node failures.

Node failure model. We model node failures by assigning each node an up-time picked uniformly at random from [0, MAXLIFETIME] where MAXLIFETIME is a simulation parameter. When a node's up-time expires, the node *resets*. That is, it disconnects from its neighbors, shuts down, and immediately rejoins the system by connecting initially to a random number of neighbors. This is similar to modeling existing nodes shutting down and leaving the system while other new nodes are simultaneously joining the system. When a node shuts down, any queries it has queued locally are dropped and resumed by the nodes that had originally generated them.[11] Finally, as nodes join and leave the system, the topology adaptation overhead is captured by the fact that each node's adaptation operations consume capacity within the node.

Figures 5 and 6 plot the CP and CP-HC, respectively, for a 10,000 node Gia system under increasing MAXLIFETIME. We see that, relative to the static case, the CP drops by approximately an order of magnitude as the MAXLIFETIME is reduced to 10.0 time units, while the hop-count rises by approximately a factor of five. Note that at a MAXLIFETIME of 10 time units, approximately 20% of the nodes reset in *every* time unit.[12] Even under this extremely stressful test, GIA's performance drops only by less than one order of magnitude. This is still an improvement of 2-4 orders of magnitude over RWRT, SUPER and FLOOD *under static conditions*.

4.8 File Downloads

The results presented above indicate that Gia can support significantly higher query loads than previously proposed approaches for distributed file searching and can maintain this performance advantage even in the face of high node churn. Gia's dramatic performance improvement stems from its unique combination of design components and its ability to funnel work to high capacity nodes in the system.

[11] The exact mechanisms for implementing query restart in a real system are discussed in Section 5.3.

[12] Compare this to typical Gnutella node life-times of 60 minutes [21].

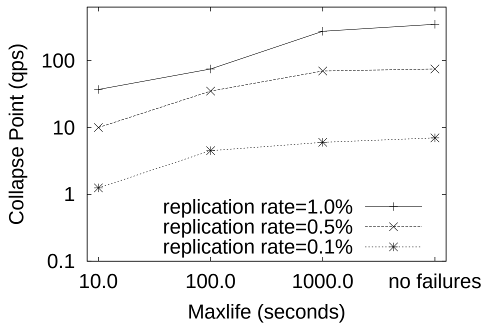

Figure 5: Collapse Point under increasing MAXLIFETIME for a 10,000 node GIA system

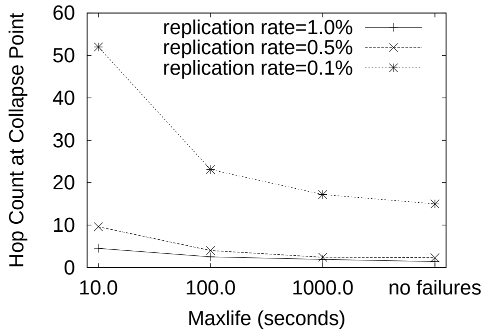

Figure 6: Hop Count under increasing MAXLIFETIME for a 10,000 node GIA system

Our results thus lead us to conclude that search in decentralized P2P systems need no longer pose insurmountable scaling problems. If so, we conjecture that the next bottleneck limiting scalability is likely to be the file download process. This will be particularly true if, as recent measurement studies indicate, file sizes continue to increase [20]. We believe that Gia's ability to harness capacity in a manner that is sensitive to the constraints of individual nodes can have a beneficial impact on downloads as well. Even as is, Gia aids downloads to the extent that users are typically directed to a high-capacity copy of a file if one exists. However this advantage is unlikely to be significant unless high capacity nodes also store more files. Thus, for Gia to be able to assist in file downloads, we would have to extend the one-hop replication used in Gia to allow the active replication of the files themselves (rather than simply pointers to files). A simple form of active replication would be for overloaded low capacity nodes to replicate popular files at the higher capacity nodes in their one-hop neighborhood. This can be done in an on-demand fashion where the high-capacity nodes replicate content only when they receive a query and a corresponding download request for that content.

To gauge the extent to which such active replication might be useful, we did a simple calculation of the total capacity of all the nodes at which a given file is available with and without this active replication scheme. The resultant numbers are listed in Table 6. We see that active replication increases the total capacity of nodes serving a given file by a factor of between 38 to 50. This appears promising, although one would need significant more analysis and simulations to validate the usefulness of this approach.

% Replication	Gia	Gia with active replication
0.1%	965	48,682
0.5%	4,716	213,922
1.0%	9,218	352,816

Table 6: Total capacity of all the nodes offering a given file with and without active replication for a 10,000 node GIA network

```
Let $C_i$ represent capacity of node $i$
if $num\_nbrs_X < min\_nbrs$ then
    return 0.0
$total \leftarrow 0.0$
for all $N \in neighbors(X)$ do
    $total \leftarrow total + \frac{C_N}{num\_nbrs_N}$
$S \leftarrow \frac{total}{C_X}$
if $S > 1.0$ or $num\_nbrs_X \geq max\_nbrs$ then
    $S \leftarrow 1.0$
return $S$
```

Algorithm 2: $satisfaction\_level(X)$
Computes how "satisfied" node X is. Returns value between 0.0 and 1.0. 1.0 $\Rightarrow$ node X is fully satisfied, while 0.0 $\Rightarrow$ it is completely dissatisfied. Values in between represent the extent of satisfaction.

5. IMPLEMENTATION AND PRACTICAL DETAILS

We implemented a prototype Gia client that incorporates all of the algorithms presented in Section 3. The client, which was written in C++, provides a command-line-based file sharing interface. In this section, we discuss some of the systems issues that our prototype implementation addresses.

5.1 Capacity settings

In our discussion so far, we have assumed that a node's capacity is a quantity that represents the number of queries that the node can handle per second. For low-bandwidth clients, query processing capacity is limited by the client's access bandwidth. On the other hand, for nodes with high-speed access connections, other issues such as the speed of the CPU, disk latencies, etc. may affect the capacity. Our prototype implementation ignores the effects of CPU speed and disk latency on query capacity. We assume that capacity is a direct function of the access bandwidth. A node can either have its end-user configure its access bandwidth (via a user interface, as is done in many Gnutella clients), or automatically determine the access bandwidth by executing a configuration script that downloads large chunks of data from well-known sites around the Internet and measures the bandwidth based upon the average time taken for the downloads. In addition, the advertised capacity of nodes can be weighted by how long the node has been in the system. This ensures that the well-connected high-capacity core of the network is composed of mostly stable nodes. In future implementations, we plan to experiment with auto-configuration scripts that take into account other factors in addition to network bandwidth and node life-times in order to determine client capacity.

5.2 Satisfaction Level: Aggressiveness of Adaptation

In Section 3.2.1, we introduced the notion of *satisfaction level* for a client. The satisfaction level determines not only whether or not to perform topology adaptation, but also how frequently it should be executed. It is a function of pre-configured min_nbrs and max_nbrs parameters, the node's current set of neighbors, their

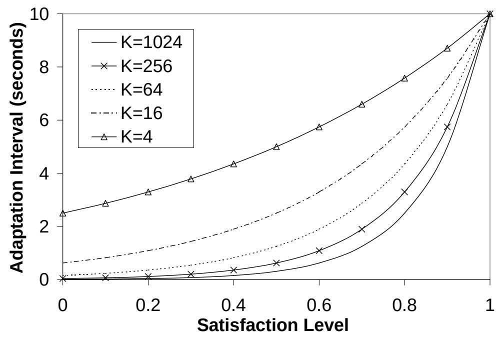

Figure 7: Adaptation Interval: A plot of the function $I = T \times K^{-(1-S)}$, where S = $satisfaction\_level()$, T = maximum interval between adaptation iterations, and K = sensitivity to $satisfaction\_level()$. In this figure, we set $T = 10seconds$ and plot curves for adaptation interval versus satisfaction level for different values of K.

capacities and their degrees. Neighbors exchange capacity information when they initially connect to each other, and periodically update each other with information about their current degree. Algorithm 2 shows the steps involved in calculating the $satisfaction\_level()$. It is essentially a measure of how close the sum of the capacities of all of a node's neighbors (normalized by their degrees) is to the node's own capacity. Thus a high capacity neighbor with a low degree contributes more to our satisfaction level than another neighbor with the same capacity but a much higher degree. The intuition behind this is that a node with capacity C will forward approximately C queries per unit time at full load and needs enough outgoing capacity from all of its neighbors to handle that load. In addition to the factors discussed above, a number of other parameters may be used to compute the satisfaction level, for example, the load on a node's neighbors, network locality, etc. However, for our prototype, we rely only on node capacity and degree to compute the satisfaction level.

The satisfaction level is key to deciding how often a node should conduct its local topology adaptation. Nodes with low satisfaction levels perform topology adaptation more frequently than satisfied nodes. We use an exponential relationship between the satisfaction level, S, and the adaptation interval, I: $I = T \times K^{-(1-S)}$, where T is the maximum interval between adaptation iterations, and K represents the aggressiveness of the adaptation. After each interval I, if a node's satisfaction level is < 1.0, it attempts to add a new neighbor. Once a node is fully satisfied, it still continues to iterate through the adaptation process, checking its satisfaction level every T seconds.

Figure 7 shows how the aggressiveness factor K affects the adaptation interval. As expected, when a node is fully satisfied ($S = 1.0$), the adaptation interval is T irrespective of the value of K. As the level of satisfaction decreases, the adaptation interval becomes shorter. For the same satisfaction level, higher values of K produce shorter intervals and hence cause a more aggressive (*i.e.*, quicker) response. In Section 5.4 we look at how the rate of topology adaptation changes in a real system with different values of K.

5.3 Query resilience

As described earlier, the Gia search protocol uses biased random walks to forward queries across the network. One of the drawbacks of using a random walk instead of flooding is that it is much more susceptible to failures in the network. If a node receives a query and dies before it can forward the query to a neighbor, that query is lost forever. This is in contrast to flooding where a query gets replicated

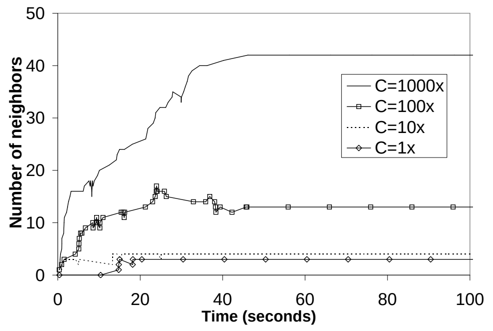

Figure 8: Progress of topology adaptation for an 83-node topology over time. The graph shows changes in the number of neighbors of four nodes (each with different capacities).

many times, and so even if a node dies without forwarding a query, there is a good chance that other copies of the query still exist in the system.

To overcome this problem, we rely on query keep-alive messages. Query responses sent back to the originator of the query act as implicit keep-alives. In addition, if a query is forwarded enough times without sending any response, we send back an explicit keep-alive message. This is implemented as a dummy query response message that has no actual matches. If the query originator does not receive any responses or keep-alive messages for a while, it can re-issue the query.

Yet another problem arises from the fact that responses are typically forwarded along the same path that the query originally arrived from. If one of the nodes in the query path dies or the topology changes due to adaptation before the responses for the query are sent back, then those responses will be dropped. When a node dies and causes responses to be dropped, the query originator will notice it because of the absence of keep-alive responses. Thus, it can reissue the query if necessary. On the other hand, topology adaptation is a controlled process and hence we can do better than wait for a timeout. When a connection is dropped as a result of a topology adaptation decision, the connection is not closed for some time later. It stops accepting any more incoming queries, but continues to forward any lingering query responses that need to be sent along that path.

5.4 Deployment

We deployed our prototype implementation on PlanetLab [16], a wide-area service deployment testbed spread across North America, Europe, Asia, and the South Pacific. To test the behavior of our algorithms in the face of diversity, we artificially introduced heterogeneity into the system by explicitly setting the capacities of the individual nodes. These experiments are by no means meant to stress all of the various components of the Gia system. We present them here as a set of early results that demonstrate the viability of this approach in an actual deployment.

We instantiated Gia clients on 83 of the PlanetLab nodes with a range of capacities. We allowed the system to run for 15 minutes before shutting down the clients. Over the course of the experiment, we tracked changes in the Gia topology to evaluate the behavior of the topology adaptation process. Figure 8 shows the changes over time to the neighborhood of each of four different nodes. These nodes were picked randomly from four capacity classes (1x, 10x, 100x, and 1000x). We notice that initially when the nodes are all "dissatisfied," they quickly gather new neighbors. The rate of topology adaptation slows down as the satisfaction level of the nodes rises and the topology eventually stabilizes to its steady state.

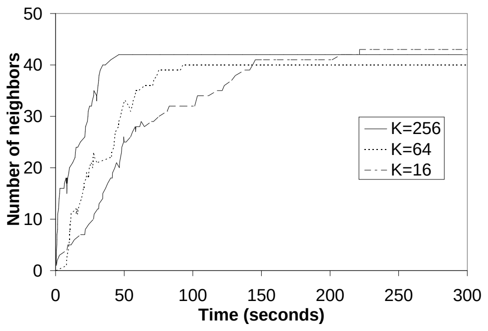

Figure 9: Progress of topology adaptation for a 1000x capacity node over time. The graph shows changes in the number of neighbors of a node over different runs of the experiment, each with a different value of K for the adaptation interval function.

In the above experiment, the 1000x capacity node takes approximately 45 seconds to reach its steady state. This time interval is closely tied to the level of aggressiveness used in the topology adaptation. Recall that the adaptation interval I is a function of the node's satisfaction level S, and its aggressiveness factor K: $I = T * K^{-(1-S)}$. In the above experiment, we set T to 10 seconds and K to 256. We ran other experiments to see how the responsiveness of the topology adaptation changes with different values of K. Figure 9 shows the behavior of a 1000x capacity node for different values of K. As can be seen from the figure, the topology adaptation does respond to changes in K, and is less aggressive when we ramp down the value of K. Thus, this parameter gives us a knob with which we can trade off the speed at which nodes attain their target topology to the rate of churn in the overall network.

6. RELATED WORK

We now look at some of the related research in this area. Since the rise and fall of Napster, many decentralized P2P systems have been proposed. Gnutella pioneered this approach, and on its footsteps many other networks such as KaZaA [23] have emerged. Although the notion of supernodes (nodes with better bandwidth connectivity) used by KaZaA and the latest versions of Gnutella helps to improve the performance of the network, it is still limited by the flooding used for communication across supernodes. Moreover, unlike Gia, the supernode approach makes just a binary decision about a node's capacity (supernode or not) and to our knowledge has no mechanisms to dynamically adapt the supernode-client topologies as the system evolves.

Numerous researchers have performed extensive measurement studies of P2P infrastructures. For example, Saroiu *et al.* [21] studied the bandwidth, latency, availability, and file sharing patterns of the nodes in Gnutella and Napster. Their study highlighted the existence of significant heterogeneity in both systems. Based on this fact, Gia is designed to accommodate heterogeneity and avoid overloading the less capable nodes in the network. Other measurement studies include [22] which shows that there exists extreme heterogeneity in the traffic volumes generated by hosts within a P2P network and that only a small fraction of hosts are stable and persist in the P2P network over long periods of time.

In addition to the work described in Section 1 [1, 12, 15], there have been other proposals for addressing the scaling problems of Gnutella. Krishnamurthy *et al.* [10] proposed a cluster-based architecture for P2P systems (CAP), which uses a network-aware clustering technique (based on a central clustering server) to group hosts into clusters. Each cluster has one or more delegate nodes that act as directory servers for the objects stored at nodes within the same cluster. In some sense, the high capacity nodes in Gia provide functionality similar to that of delegate nodes. However, unlike CAP, Gia adapts its topology to cluster around high-capacity nodes in a fully decentralized manner and explicitly takes node capacity into account in all facets of its design.

7. CONCLUSION

We have proposed modifying Gnutella's algorithms to include flow control, dynamic topology adaptation, one-hop replication, and careful attention to node heterogeneity. Our simulations suggest that these modifications provide three to five orders of magnitude improvement in the total capacity of the system while retaining significant robustness to failures. The increased capacity is not due to any single design innovation, but is the result of the synergy of the combination of all of the modifications. While making search much more scalable, the design also has potential to improve the system's download capacity by more fully distributing the load. Thus, a few simple changes to Gnutella's search operations would result in dramatic improvements in its scalability.

Why is this result interesting? The most plausible alternative to Gia is a DHT-based design. As we argued in Section 2, we believe that DHTs, while more efficient at many tasks, are not well suited for mass-market file sharing. In particular, their ability to find *needles*, *i.e.*, exceedingly rare files, is not needed in a mass-market file-sharing environment, while their ability to efficiently implement keyword search, which is crucial for this application, is still unproven.

Another alternative, perhaps too readily dismissed by the research community, is that of centralized search as in the original Napster. The reflex response from the research community is that such designs are inherently unscalable, but the examples of Google, Yahoo, and other large sites tell us that scalability does not pose an insurmountable hurdle. In reality, the real barriers to Napster-like designs are not technical but legal and financial. The demise of Napster is due to it being used for unauthorized exchanges of copyrighted material. Adopting decentralized designs merely to avoid prosecution for such acts is hardly a noble exercise. From a financial standpoint, while scaling a centralized search site is technically feasible, it requires a sizeable capital investment in the infrastructure. Thus, this approach can only be adopted when there is an underlying business model to the application. In contrast, decentralized designs need no large infrastructure expenditures.

Thus, we view our work not as facilitating copyright avoidance but as enabling the sharing of files in cases where there is no underlying business model. This is what the web did for publishing—allowing any author access to a large audience regardless of commercial viability—and we hope to support a similar ability in file sharing.

8. REFERENCES

[1] ADAMIC, L. A., LUKOSE, R. M., PUNIYANI, A. R., AND HUBERMAN, B. A. Search in Power-law Networks. *Physical Review E 64* (2001).

[2] ADAR, E., AND HUBERMAN, B. A. Free Riding on Gnutella. *First Monday, Internet Journal* (Oct. 2000). Available at `http://www.firstmonday.dk/issues/issue5_10/adar/index.html`.

[3] BHAGWAN, R., SAVAGE, S., AND VOELKER, G. Understanding Availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03).* (Berkeley, CA, Feb. 2003).

[4] C—NET NEWS. Napster among fastest-growing Net technologies, Oct. 2000. `http://news.com.com/2100-1023-246648.html`.

[5] GNUCLEUS. The Gnutella Web Caching System, 2002. `http://www.gnucleus.net/gwebcache/`.

[6] GNUTELLA DEVELOPMENT FORUM. The Gnutella v0.6 Protocol, 2001. `http://groups.yahoo.com/group/the_gdf/files/`.

[7] GNUTELLA DEVELOPMENT FORUM. The Gnutella Ultrapeer Proposal, 2002. `http://groups.yahoo.com/group/the_gdf/files/Proposals/Ultrapeer/`.

[8] GNUTELLA.WEGO.COM. Gnutella: Distributed Information Sharing, 2000. `http://gnutella.wego.com/`.

[9] GOYAL, P., VIN, H. M., AND CHENG, H. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM '96* (Stanford, CA, Aug. 1996).

[10] KRISHNAMURTHY, B., WANG, J., AND XIE, Y. Early Measurements of a Cluster-based Architecture for P2P Systems. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2001* (San Francisco, CA, Nov. 2001).

[11] LI, J., LOO, B. T., HELLERSTEIN, J., KAASHOEK, F., KARGER, D. R., AND MORRIS, R. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03).* (Berkeley, CA, Feb. 2003).

[12] LV, Q., CAO, P., COHEN, E., LI, K., AND SHENKER, S. Search and Replication in Unstructured Peer-to-Peer Networks . In *Proceedings of 16th ACM International Conference on Supercomputing(ICS'02)* (New York, NY, June 2002).

[13] LV, Q., RATNASAMY, S., AND SHENKER, S. Can Heterogeneity Make Gnutella Scalable. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '03).* (Cambridge, MA, Mar. 2002).

[14] METAMACHINE. The Overnet File-sharing Network, 2002. `http://www.overnet.com/`.

[15] OSOKINE, S. The Flow Control Algorithm for the Distributed 'Broadcast-Route' Networks with Reliable Transport Links., Jan. 2001. `http://www.grouter.net/gnutella/flowcntl.htm`.

[16] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the ACM HotNets-I Workshop* (Princeton, NJ, Oct. 2002). See also `http://www.planet-lab.org/`.

[17] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content-addressable Network. In *Proceedings of ACM SIGCOMM 2001* (San Diego, CA, Aug. 2001).

[18] REYNOLDS, P., AND VAHDAT, A. Effi cient Peer-to-Peer Keyword Searching. Technical report, Duke University, Durham, NC, 2002. Available at `http://issg.cs.duke.edu/search/`.

[19] RIPEANU, M., FOSTER, I., AND IAMNITCHI, A. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *IEEE Internet Computing Journal 6*, 1 (2002).

[20] SAROIU, S., GUMMADI, K. P., DUNN, R. J., GRIBBLE, S. D., AND LEVY, H. M. An Analysis of Internet Content Delivery Systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)* (Boston, MA, Dec. 2002).

[21] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02)* (San Jose, CA, Jan. 2002).

[22] SEN, S., AND WANG, J. Analyzing Peer-to-Peer Traffi c Across Large Networks. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop 2002* (Marseille, France, Nov. 2002).

[23] SHARMAN NETWORKS LTD. KaZaA Media Desktop, 2001. `http://www.kazaa.com/`.

[24] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM 2001* (San Diego, CA, Aug. 2001).

[25] TANG, C., XU, Z., AND MAHALINGAM, M. pSearch: Information Retrieval in Structured Overlays. In *Proceedings of the ACM HotNets-I Workshop* (Princeton, NJ, Oct. 2002).

[26] ZHAO, B., KUBIATOWICZ, J., AND JOSEPH, A. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Tech. rep., University of California, Berkeley, 2001.