Ad hoc, self-supervising peer-to-peer search networks

BRIAN F. COOPER
Georgia Institute of Technology
and
HECTOR GARCIA-MOLINA
Stanford University

Peer-to-peer search networks are a popular and widely deployed means of searching massively distributed digital information repositories. Unfortunately, as such networks grow, peers may become overloaded processing messages from other peers. This paper examines how to reduce the load on nodes in P2P networks by allowing them to self-organize into a relatively efficient network, and then self-tune to make the network even more efficient. Two local operations used by a peer are introduced: **connect()**, in which the peer forms an ad hoc search or index link to another peer, and **break()**, in which the peer breaks a link that is producing too much load. By replacing fixed rules with dynamic local decision-making, such "self-supervising" networks can better adjust to network conditions. Different ways to implement **connect()** and **break()** are described, and the network structures that form under different configurations are examined. Simulation results indicate that the ad hoc networks formed using the described techniques are more efficient than popular supernode topologies for several important scenarios. Results for the fault tolerance and search latency of such ad hoc networks are also presented.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*; *information networks*; *performance evaluation*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*; C.2.4 [**Computer-Communication Networks**]: Distributed systems—*Distributed databases*; *distributed applications*

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Peer-to-peer systems, information search and discovery

1. INTRODUCTION

Peer-to-peer search networks are an effective mechanism for sharing information between large numbers of users. Existing networks such as Kazaa support several

Author's addresses: B. F. Cooper, College of Computing, Georgia Institute of Technology, 801 Atlantic Dr., Atlanta, GA 30332, `cooperb@cc.gatech.edu`
H. Garcia-Molina, Department of Computer Science, Stanford University, Gates Hall Room 434, Stanford, CA 94305, `hector@db.stanford.edu`
This material is based upon work supported by the National Science Foundation under Award 9811992.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

million simultaneous users and allow those users to search and retrieve data from a distributed repository containing hundreds of millions of objects and multiple petabytes of data. The power of the system comes from aggregating the resources of millions of computers, even though each computer may be no more powerful than a desktop computer.

Despite the popularity of peer-to-peer search services, they are still quite inefficient. The search service can place a large load on both the peers and the network connecting them. The flooding nature of queries in many systems means that many peers are impacted whenever new peers join. The heavy load imposed by the network limits the scalability of the system, and may persuade users or ISPs to limit the resources they contribute or may convince users not to join at all.

Recently, a great deal of research has been focused on improving the efficiency and scalability of search networks, so that they can be used for tasks beyond the traditional role of multimedia file-sharing. Most of the proposed solutions have fixed rules for the network topology. For example, in supernode networks, "normal" nodes can only connect to supernodes, while in distributed hash tables such as Chord [Stoica et al. 2001] or CAN [Ratnasamy et al. 2001], a node's neighbors are a fixed function of the node's id. These fixed rules also specify what information is exchanged between peers. For example, in supernode networks, supernodes index the content of normal nodes but do not send indexing information to other supernodes. While each of these solutions have certain advantages, we wanted to explore the possibility of constructing search networks in a more flexible and ad hoc manner. In particular, we wanted to see if we could make the network adaptive to form an efficient topology on its own, rather than specifying a particular topology beforehand.

In our approach, a node can connect to any other node in the network, using a **connect()** operation. If a node becomes overloaded because it has too many neighbors, it uses a **break()** operation to drop some connections. The disconnected nodes may then re-**connect()** to other nodes. This process of connecting and breaking links replaces the fixed topology rules of previous solutions, and allows the network to self-organize and self-tune to form an efficient topology.

Another feature of our approach is that connections can be search links, in which search information is sent, or index links, in which indexing information is sent. When sending indexing information to their neighbors, nodes shed load by asking their neighbors to take on some of the searching work. In previous work, a node takes on indexing as a fixed function of its role in the topology. Our approach is to allow indexes to be created and disseminated dynamically in a way that best improves the efficiency of the network.

Topology adaptation for improving efficiency has been studied before [Lv et al. 2002]. However, our approach differs in several ways. One difference is that we treat index links as first-class objects, to be created explicitly in order to reduce overall load. Moreover, in our system nodes are only responsible for tracking their own load and capacities, and do not have to track the capacities of their neighbors in order to effect topology rearrangements. A detailed comparison with this and other previous work is presented in Section 5.

Our target is content discovery networks like Gnutella [gnu 2003; Ripeanu and

Foster 2002] or Kazaa [kaz 2003], where keywords are used to locate content. These networks represent the most popular and widely deployed peer-to-peer systems, and thus it is worth investigating how to optimize them. Distributed hash tables focus on a different problem: locating an object by unique id, rather than by keyword. It may be possible to adapt our techniques for distributed hashing, although we have not yet done so. Also, we are focusing in this paper on optimizing networks through dynamic topology adaptation. Other optimizations, such as time-to-live stopping rules, random walks [Lv et al. 2002], or proactive content replication [Cohen and Shenker 2002] can still be used in the topologies resulting from our techniques, although study is required to quantify their effect.

Our results indicate that ad hoc topology adaptation results in significant efficiency gains in several situations. In particular, ad hoc networks reduce load on nodes by more than 30 percent when compared to supernode networks (the most popular existing peer-to-peer networks). Moreover, in a supernode network all of the work is done by a few very powerful supernodes. In some applications (such as a network of digital archives), all nodes may have similar capabilities and there may not be nodes powerful enough to serve as supernodes. Ad hoc networks can be tuned for this scenario as well, reducing load on the most loaded nodes by over 80 percent while only increasing the load on "normal" nodes by a factor of 1.2 compared to supernode networks. These results illustrate the main strength of ad hoc techniques: the network rearranges its topology to best fit the needs and load pattern of the search service.

In this paper, we study self-organizing and self-tuning search networks, which we call "ad hoc, self-supervising" networks. Specifically, we make the following contributions:

—We discuss how to build peer-to-peer networks in an ad hoc way from search links and index links, using the **connect()** and **break()** operations.

—We examine and evaluate alternatives for implementing the **connect()** and **break()** operations.

—We present simulation experiments that show how to best construct an ad hoc self-supervising network, and that compare such networks to supernode networks, a popular existing architecture. Our results show that in many cases an ad hoc network is more efficient than a supernode network, while providing good fault tolerance and low search latency.

This paper is organized is follows. In Section 2, we discuss our model of peer-to-peer search networks. In Section 3 we present the basic techniques for dynamically constructing search networks in a self-supervising manner. In Section 4 we examine evaluation results for our techniques. In Section 5 we review related work and in Section 6 we present our conclusions.

2. PEER-TO-PEER SEARCH NETWORKS

Nodes in a P2P search network collaborate to provide search and retrieval of digital documents. When a user wishes to find an object, he submits a query to a node, which attempts to answer the query as best it can. The node then forwards the query to other nodes, who also attempt to answer the query. Whenever a node

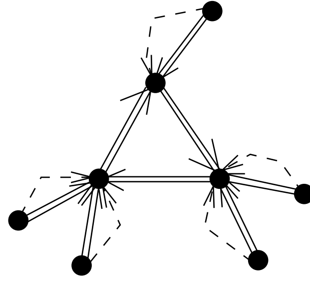


Fig. 1. A supernode network.

finds an object matching the query, a result is returned to the user indicating the location of the object. The user can then directly retrieve the object from the node holding the object.

Nodes in a search network form a partially-connected overlay over a fully connected network such as the Internet. In our model, there are two types of connections in this overlay network:

—*Search links* ($X \Longrightarrow Y$), which are used to forward queries between nodes

—*Index links* ($X \dashrightarrow Y$), which are used to send copies of content indexes between nodes

These connections are uni-directional; of course, a bi-directional link can be modeled as a pair of uni-directional links. An example of a network modeled in this way is shown in Figure 1. This network is a supernode network, used in systems such as Kazaa. The central nodes are supernodes, and the other nodes send their indexes to a supernode over index links. Nodes send queries to their supernode over search links. These queries get processed at the supernode and then forwarded to other supernodes for more processing. Because there are no links directed at normal nodes (e.g., non-supernodes), normal nodes only handle the load they themselves generate.

With an index link, a node sends indexing information to a neighbor. For example, in order to aid in processing searches, each node constructs an inverted list of words in the content of its digital objects, a list of the object titles, or some other index. If there is an index link $B \dashrightarrow A$, then the node B sends its index to node A. Now, when node A receives a query, it can process that query over its own content, and, using B's index, over B's content. Index links reduce the need to flood the entire network with queries, since node A can perform searching operations over node B's repository without B needing to process the query. Note that A does not have to have a copy of B's content; A instead has index entries that support searches over B's content.

Search links form a forwarding network: when a node receives a search message, that node processes the message and then forwards the message on other search links. In contrast, index messages are not forwarded beyond a single link. In general, there could be forwarding index links or non-forwarding search links; see [Cooper and Garcia-Molina 2003a] for details.

3. CONSTRUCTING SEARCH NETWORKS USING LOCAL DECISIONS

Let us now see how an ad hoc search network can be constructed. Imagine that there is a node, A, that wishes to join a search network. First, A must discover some nodes that are already in the network to serve as A's neighbors. This is usually done using a *hostcatcher*, which is a node that tracks which other nodes are currently in the network. Node A must contact a hostcatcher at a well-known address, and get some node ids. Later, when A has been in the network for a while, it may want to keep its own list of live node ids to avoid depending on the centralized hostcatcher. One mechanism for tracking node ids in a distributed manner is for each node to cache the ids of nodes gathered from search messages, search responses, or Gnutella-style "ping/pong" messages. Nodes can then use this "pong-cache" [Yang et al. 2004] to find new nodes for making connections.

Now that A knows about some potential neighbors, it performs the **connect()** operation to form links. For example, A may select B as a potential neighbor. Node A may decide, say, to form a search link $A \Longrightarrow B$ and an index link $A \dashrightarrow B$. If B agrees to these links, then the connections are made and A begins sending update messages to B and sending and forwarding search messages to B. Node A may also decide to **connect()** to another node, C. In this case, A may simply decide to form an index link $A \dashleftarrow C$. Since this link is directed from C to A, A will have a copy of C's index and can search C's content. This process continues, with A connecting to neighbors until it feels it has enough connections (for example, more than some parameter m). As other nodes (D, E ...) join the network, they will also perform **connect()**, possibly connecting to A.

At some point, node A may become overloaded with search and update messages. This may happen because other nodes have connected directly to A, or because other nodes have connected elsewhere in the network but their search messages are being forwarded to A. At this point, A can shed load by simply dropping some links. This is called the **break()** operation. For example, A may decide to disconnect a link $A \Longleftarrow F$, because A is receiving the most search messages from that link. Nodes that have been disconnected from A (such as F) can now perform **connect()** to other nodes to replace the broken connection to A.

Using **connect()**, nodes self-organize into a search network. Then, using **break()**, the network becomes self-tuning: overloaded nodes shed load and other nodes pick up the slack. Using **connect()** and **break()**, nodes rearrange the topology to share load effectively. Moreover, if search load predominates, nodes can drop search links and replace them with index links. Similarly, nodes can replace index links with search links if update load predominates.

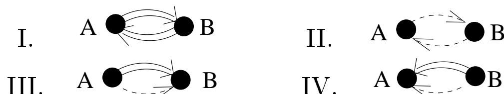
3.1 The **connect()** operation

In our framework, there is no pre-determination of how two nodes will connect. In the simplest case, a node A could simply create a single link with another node B. Such a link could be a search link or index link, and may be directed "forward" ($A \rightarrow B$) or "backward" ($A \leftarrow B$). We call this a "one-way **connect()**" operation, since a single link is formed in one direction. Multiple one-way **connect()** operations could result in multiple links between a pair of nodes. Note that multiple links between the same nodes, while conceptually distinct, could be multiplexed over a

single TCP connection or, alternatively, implemented over connectionless UDP.

When performing a one-way **connect()**, a node must decide which link to form with a particular neighbor, and the easiest thing to do is to randomly choose the link type and direction. This process is described by two parameters: P_{sl}, the probability that a search link is chosen instead of an index link, and P_f, the probability that the link is a forward link. For example, node A may set $P_f = 1$ and $P_{sl} = 0.5$. Then, whenever A performs a **connect()**, it would form a forward link, but would flip a coin to determine whether to form a search link or index link. Nodes may actually use any decision criteria they like for forming links. For example, node A may prefer to form links to nodes that would not otherwise be searchable by A. However, for simplicity we will treat link formation as a random process governed by the P_{sl} and P_f parameters.

One-way connections have the disadvantage of being asymmetric; out of the pair of nodes, only one node can search the other unless multiple **connect()**s are performed. A more symmetric method of connecting is to form two links in one **connect()** operation in order to ensure that both connected nodes can search each other. We call this method "two-way **connect()**." There are four combinations of two links which ensure that the connected nodes can search each other:



In type I, A and B search each other directly, while in type II A and B search each other indirectly. In types III and IV, one node searches the other directly and is itself searched indirectly.

A node must decide between the four connection types (I, II, III and IV) when performing a two-way **connect()**. As with one-way **connect()**s, the node can assign probabilities to each possibility, and then choose randomly. For example, a node that uses $P_I = 0.5$, $P_{II} = 0.5$, $P_{III} = 0$ and $P_{IV} = 0$ would form type I and type II links with equal probability but would never form type III or type IV links.

The probability parameters P_{sl}, P_f, P_I, P_{II} and so on can be set in a number of ways. Our approach is to run experiments to find values that tend to produce efficient networks in many cases; the results are presented in Section 4. Then, the "best" values would be agreed upon as part of the network protocol. For example, a protocol with $P_I = 0.5$ and $P_{II} = 0.5$ would allow only type I and II links, while a protocol with $P_I = 1$ would create only type I links (as in Gnutella). It may be possible for each node to learn values for these parameters that are individually best for that node. While this may improve the load for that node, it is not clear if the whole network would benefit from such local optimization. Furthermore, it is difficult to study this scenario because nodes would have so many degrees of freedom (when to create links, when to break links, the links to create or break, the setting of the probability parameters). In order to make our analysis tractable, we will assume that all nodes in the network use the same setting for the parameters, fixing that degree of freedom so that we can study the other choices a node must make. Then, we can examine adaptive parameter setting as part of our future work.

There are other ways of connecting besides one-way and two-way **connect()**

(for example, a **connect()** that forms A B) that may also be interesting to study. However, here we will restrict our attention to one-way and two-way connections to simplify our analysis.

We have designed **connect()** as a randomized process for several reasons. First, the **connect()** operation is lightweight and easy to implement. Nodes do not need to guess which links will be best at any given time, or to predict future traffic patterns. Instead, they make some random links, and then later, **break()** the inefficient links based on actual traffic behavior. Second, a randomized process helps to make the network more robust by creating a mixture of different link types. In a deterministic process, all of the nodes might decide to create search links because they appear to be locally best, when creating a few index links would have been better for the global efficiency. It still may be useful to favor one type of link in certain scenarios, but this can be handled in our model by choosing appropriate values for the probability parameters. For example, in Section 4, we discuss experimental results which show that one parameter setting is best if nodes have roughly equal capacities, while another setting is best for nodes that have varying capabilities.

Because **connect()** is non-deterministic, it is possible that the resulting network may be partitioned, just as in existing networks such as Gnutella or supernode networks. However, in practice, if each node creates even a small number of links, partitions are rare. For example, experiments in [Cooper and Garcia-Molina 2003b] show only four search links per node are needed to avoid partitions.

3.2 Propertied **connect()**

In its simplest form, **connect()** makes connections randomly, without aiming for a particular topology. While this method produces a flexible network in which each node connects as it sees fit, it may also lead to inefficient networks, since adding a new link results in higher load on one or more nodes but may not improve the effectiveness of the network. Consider for example the network in Figure 2a. In this network, node A is able to search node B because there is a search path from A to B. Now consider Figure 2b. In this network, an index link has been added from B to A. This index link results in added load on node A, since A must now process index updates from B. However, the extra link does not benefit A, since A was already able to search B via a search path. In fact, in the network of Figure 2b, the index link does not increase the number of searchable nodes for any node. We say that the link is *redundant*.

More formally, we define *redundancy* as a property of a network where a link can be removed without reducing the *coverage* for any node. The *coverage* of a node A is the number of other nodes that A can search. Redundancy causes extra load on nodes, although redundancy may be useful for improving the fault tolerance of the network or reducing the time for searches to complete. Here, our main focus is improving the scalability of the system by reducing load, so we will examine how to reduce load by avoiding redundancy.

Two topological features that exhibit redundancy are the *one-index-cycle* and the *search-fork*. A one-index-cycle is a cycle that includes a single index link. An example is shown in Figure 2b. A search-fork is a triangle-shaped feature, in which

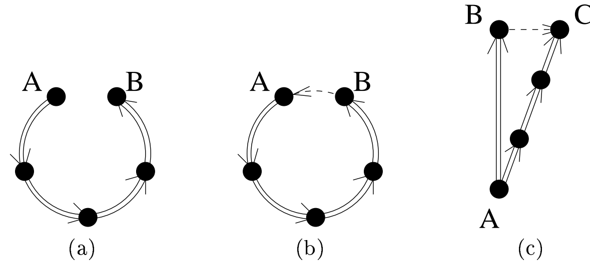


Fig. 2. Networks: (a) search path, (b) one-index-cycle, (c) search-fork

there is a search link from A to B, a search path from A to C and an index link from B to C. An example is shown in Figure 2c, where the redundant link is the search link from A to B. This feature leads to unnecessary load because B must process A's searches even though those searches will also be processed over B's index by C.

We can improve the effectiveness of the **connect()** operation by specifying that a **connect()** can only complete if it does not form a one-index-cycle or a search-fork. Although this principle does not guarantee that the resulting network is efficient, it does eliminate some clear sources of inefficiency. We refer to a **connect()** operation that avoids one-index-cycle and search-fork features as a "propertied **connect()**," that is, a **connect()** that avoids introducing bad properties into the network.

In order to perform a propertied **connect()**, a node must be able to detect whether a link that it is about to make will create a search-fork or a one-index-cycle. It is possible to detect these properties without adding too much overhead to the network if we extend the ping-pong mechanism of P2P networks. In Gnutella, a node periodically announces its presence with a ping message, and other nodes respond with pongs. If a node "hears" its own pings, there must be a cycle in the network. By adding counters to the ping message, the message can act as a simple state machine that can detect specific features, like one-index-cycles and search forks. Whenever a node forwards a ping message on a link, it simply updates the counter corresponding to the link type. Then, a node can perform a propertied **connect()** by tentatively forming a link, listening for its own pings to see if that link created a search-fork or one-index-cycle, and breaking the link if it has.

Note also that avoiding search-forks and one-index-cycles is not sufficient to ensure that no redundancy exists in the network. Elsewhere [Cooper and Garcia-Molina 2003a], we have shown that two other properties, index-forks and search-loops, must also be avoided to eliminate all redundancy. We experimented with avoiding all four properties on various networks, but found that such networks tended to have low coverage, as many potential links were rejected. Avoiding search-forks and one-index cycles did not impact coverage as much, while still significantly reducing the load on network nodes. Therefore, we found it was useful to detect and avoid just the one-index-cycle and search-fork properties.

3.3 The **break()** operation

When a node becomes overloaded, there are several ways that it can **break()** links. Each method depends on a parameter called the *break threshold* B_T, which

determines what load level constitutes "overloaded."

—*MostLoadedLink*: the link causing the most load is broken, if that link is transmitting more than B_T messages per unit time.

—*MostLoadedLinks*: all links transmitting more than B_T messages per unit time are broken.

—*MostLoadedType*: if the majority of the load on a node is search load, and the total search load on the node larger than B_T, then all search links are broken. In contrast, if most of the load is update load, and the total update load is larger than B_T, then all index links are broken.

—*MostLoadedLinkOfType*: if the majority of the load on a node is search load, and the search load is larger than B_T, then the most heavily loaded search link is broken. In contrast, if the majority of the load on a node is update load, and the update load is larger than B_T, then the most heavily loaded index link is broken.

The goal of the MostLoadedLink and MostLoadedLinks methods is to simply break whatever links are causing a node A to be overloaded. Hopefully, the disconnected neighbors will reconnect to nodes other than A, so that the load is better spread around the network. The MostLoadedType and MostLoadedLinkOfType methods are more focused on determining whether it is searching or updating that is causing the node to be overloaded, and then breaking specific links to reduce the search or update load as appropriate. Then, a network that has a large search load will increase the number of index links and reduce the number of search links, self-tuning itself to reduce search load. At the same time, an index-heavy network with a large update load will self-tune to replace index links with search links and reduce the update load.

A node may monitor load and break links continuously. For example, a node may have a threshold B_T and whenever a link starts to produce more load than B_T, that link is broken. In contrast, a node may perform breaking periodically, say, every few minutes. Whenever it is time to perform a **break()**, the node determines which links, if any, are above the threshold B_T, and breaks them. We can label the break interval B_I. In the periodic case, a node may set B_T to zero. For example, if the node was using the MostLoadedLink method with $B_T = 0$, whenever it was time to **break()**, the most loaded link would be broken, regardless of how much load it carried. In experiments in Section 4, we examine good values of B_I and B_T. As with the probability parameters of Section 3.1, we assume for now that these parameters are fixed as part of the protocol.

4. EVALUATION

We have evaluated our techniques by simulating network construction and analyzing the properties of the resulting networks. We chose simulation as our methodology so that we could test a wide variety of network parameters and resulting topologies. The simulation setup is described in Section 4.1. Our first task was to identify how best to tune the **connect()** and **break()** operations. In Section 3 we presented several alternatives, and these options are summarized in Table I. For example, a node may perform propertied or non-propertied **connect()**s. In either case, the node may make one-way or two-way connections. For each of these connection

Table I. Tuning **connect()** and **break()**.

connect() parameters		
Type	propertied or non-propertied	
Connection	one-way or two-way	
Link probabilities	one-way	P_f, P_{sl}
	two-way	P_I, P_{II}, P_{III}, P_{IV}
break() parameters		
Method	MostLoadedLink, MostLoadedLinks, MostLoadedType, MostLoadedLinkOfType	
Threshold	B_T	
Interval	B_I	

types, there are various probability parameters that determine which specific links are made. Similarly, there are many options when performing a **break()** operation.

Once we know which parameters work well for **connect()** and **break()**, we can compare the resulting ad hoc networks to existing techniques. We chose to compare our techniques to the two most popular deployed network types, pure search (e.g. Gnutella) and supernode (e.g. Kazaa) networks. These networks provide the same search semantics as the ad hoc self-supervising networks we are studying here. Recently, researchers have proposed other types of networks, such as distributed hash tables and random-walk search networks. However, as these new network types are not yet widely deployed, and as they provide different search semantics than our ad hoc networks (see Section 5), we felt it was most appropriate to conduct an "apples-to-apples" comparison to the widely used and quite popular supernode and pure search network types.

The efficiency metric we used for our evaluation is *messages per covered node* (MCN): the number of messages per unit time processed by a node, divided by the coverage for that node. For example, if node A has a search load of 15 messages/unit time and an update load of 7 messages/unit time, and can search 11 nodes directly or indirectly, then the MCN for that node is $(7 + 15)/11 = 2$ messages/covered node. We chose MCN as our metric because it represents the amount of processing and bandwidth resources a node must contribute for each node it is able to search, and thus effectively measures the inherent efficiency of the network regardless of the network size or coverage. That is, MCN allows us to optimize for structures that give the most coverage at the least cost. In our experiments, we calculate the MCN for each node, and then take the average MCN of all nodes in the network.

We also require in our experiments that coverage be relatively high, specifically that coverage is greater than forty percent of the network. If a particular technique results in a network with low MCN but also low coverage, we reject it as the network, though "efficient," is not providing service to member peers. Coverage does not have to be 100 percent, as existing networks such as Gnutella and Kazaa themselves do not provide full coverage. Our reasoning is that as long as the network provides "enough" coverage so that peers can find content, then we can turn our attention to efficiency by minimizing MCN.

Table II. Simulation parameters.

Param	*Description*	*Base values*
N	Nodes per network	200
R	Runs per experiment	10
L_S, L_U	Mean search and update messages per unit time per node (Normally distributed with $\sigma = \frac{1}{4}$mean)	$L_S = L_U/10$, $L_S = L_U$, $L_S = 10 \times L_U$
L_{tot}	$L_S + L_U$	100
M	Minimum desired links per node	20
I_B	Mean interval between node births	10 ticks
μ_L	Mean node lifetime	1000 ticks
σ_L	Node lifetime standard deviation	250 ticks

4.1 Simulation setup

Our model of P2P search networks is simple, yet it is powerful enough to provide interesting insights into the behavior of peer-to-peer search networks. Similarly, we make simplifying assumptions in our simulation model so that we can study the inherent properties of a wide range of network types and parameter configurations. The key parameters for our simulations and their base values are shown in Table II; these parameters are discussed in detail below. The base values in Table II indicate the values used in the results we report. We tried a variety of values for these parameters, and the overall results remained consistent except where noted. (We present some results on the scalability of our techniques to larger networks in Section 4.5.)

In our simulations, we construct a model of each network, containing nodes and search and index links. Initially, the network is empty, and nodes are "born" at random intervals, on average every I_B time ticks. When a node is born, it selects random nodes already in the network and **connect()**s to those nodes, until the new node has at least M incoming or outgoing links. If the node is unable to form M links, say because there are not enough nodes in the network, then that nodes waits a few simulation time ticks and tries again. Once a node A has at least M links, the node does not perform **connect()** unless its links have dropped below M, although other nodes may connect to A bringing the number of A's links above M.

In our experiments we used $M = 20$. Since there may be one to four links between two nodes, $M = 20$ represents between 5 to 20 neighbors for a node. We experimented with a range of values for M, and found that $M = 20$ results in networks with relatively high coverage, even as the number of nodes in the network N grows. Moreover, it is reasonable to expect a node to have 20 open connections, as we have observed machines running a Gnutella client with that many open Gnutella connections. At the same time, our results showed that M affects coverage (higher M increases coverage) but does not change the basic results and conclusions we report below.

For the **break()** operation, the simulation schedules "break" events every B_I

simulation time ticks. A break event involves a two step process: first, every node calculates its local load and decides which, if any, links to break, and second, the selected links are removed from the network. This represents a simplified version of the **break()** operation in a real network, where nodes would continously monitor their load and perform a **break()** whenever they felt it necessary. By implementing breaks in a synchronous manner, where all breaks occur at once, we were able to create a more controlled simulation environment to effectively study such issues as the impact of the break frequency on the efficiency of the network.

We also tested two scenarios, one where nodes joined the network and stayed until the end of the simulation, and another scenario where nodes were given a "lifetime" value, and when the simulation reached the birthday plus the lifetime for a given node, that node and all of its associated links were removed from the network. Lifetime values were chosen randomly from a normal distribution with a mean of μ_L and standard deviation σ_L. Note that in the case where nodes leave the network, only about half the nodes were "alive" at any given time in the simulation. In either scenario, we stopped the simulation after the last node to be born performed its birthday **connect()**s, and calculated our load metric. In both scenarios (nodes leave or do not leave) results were roughly equivalent; although the absolute value of our measurements may change, our conclusions remain valid, unless explicitly stated.

In the case of propertied **connect()**s, we only allowed a link to be added to the network if it did not create a one-index-cycle or search-fork. Since we are primarily concerned here with measuring the potential benefits of propertied **connect()**s, we assumed that the network being constructed had some mechanism for detecting properties.

We simulated load patterns by assigning to each node two values: L_S, the number of messages generated on search links by the node per unit time, and L_U, the number of index update messages sent out on index links per unit time. We studied scenarios where the mean L_S was much higher than, equal to, and much less than the mean L_U. For simplicity, we assume that both search and update messages are equally expensive to process. Certainly, one type of message may be more expensive than another, but this situation is analogous in our framework to a situation where there are more messages of one type. The load on a node A is the total number of search and update messages processed by that node, e.g. the sum of the L_U values for nodes with an index link to A and the L_S values for the nodes with a search path to A.

We tested networks built with one-way **connect()** operations with various values of P_{sl} and P_f. In our discussion and figures below, it is convenient to adopt a shorthand for naming a network type depending on the values of these parameters: "1-way(fP_f/sP_{sl})." For example, "1-way(f1.0/s0.5)" indicates that a network was built with one-way **connect()** where $P_f = 1$ and $P_{sl} = 0.5$. Similarly, we tested networks built with two-way **connect()** operations with varying values of $P_I ... P_{IV}$. In this case, a convenient shorthand for naming network types is to list just the connection types that were used; for example "2-way(I/II)" or "2-way(I/III/IV)". For each type of two-way network, we assigned equal probabilities to the different connection types used in that network. Thus, in a "2-way(I/II)" network, $P_I =$

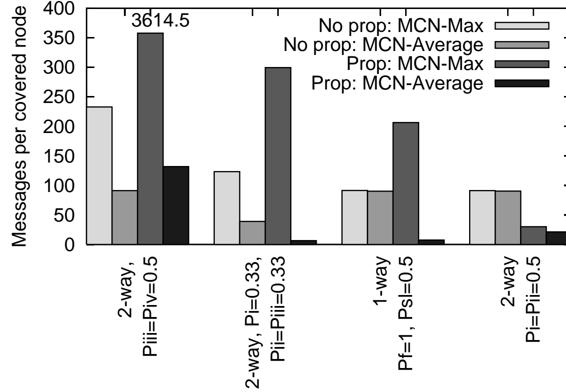


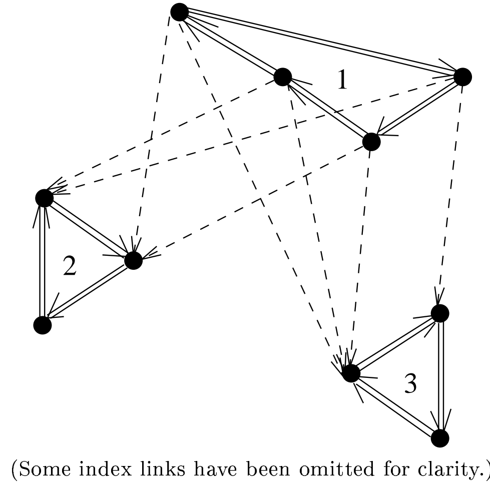
Fig. 3. Non-propertied **connect()** versus propertied **connect()**.

$P_{II} = 0.5$ while $P_{III} = P_{IV} = 0$.

4.2 The **connect()** operation

We start by studying the **connect()** operation. For now, nodes do not perform **break()**s, so that we can understand and tune the behavior of **connect()**. We ran an experiment where nodes joined the network using non-propertied **connect()**s, and compared the resulting load to that in networks built with propertied **connect()**s. We ran this experiment for 28 different network types, and four representative results from the scenario where $L_S >> L_U$ are shown in Figure 3. The horizontal axis shows the type of network configuration, while the vertical axis shows the messages per covered node (MCN) metric. We show both MCN-Average, which represents the average MCN over all nodes in the network, and MCN-Max, which represents the load for the node with the highest MCN in the network. The figure includes MCN-Average and MCN-Max for both the non-propertied **connect()** case ("No prop") and the propertied **connect()** case ("Prop"). Note that one bar, "Prop: MCN-Max" for 2-way(III/IV) is so high that to make the graph readable we had to chop off the bar and show the actual value instead ("3614.5").

This graph shows three different effects of the propertied **connect()** operation: sometimes the effect is beneficial, sometimes it is detrimental, and sometimes the results are mixed, as MCN-Max increases but MCN-Average decreases. An example of where property checking is detrimental is the 2-way(III/IV) network type shown in Figure 3. A large increase in MCN-Max was also observed in several other networks that had type III or type IV links, which are both search link/index link pairs. When a node A has an incoming search/index pair to a node B, it cannot form outgoing links to B or B's ancestors, since such outgoing links would form a one-index-cycle. The result is that the FSL/NIL pairs impose a sort of partial ordering on nodes, with some nodes becoming heavily loaded termini at the end of long chains of search/index pairs. The same effect is observed to a lesser degree in many one-way networks, where the strictness of the partial ordering is



(Some index links have been omitted for clarity.)

Fig. 4. A "search cluster" topology resulting from 2-way(I/II) **connect()**.

lessened by the fact that single links, and not search/index pairs are formed. The large MCN-Max pulls up MCN-Average, sometimes overcoming the benefits of property checking (as in the case of 2-way(III/IV)) and sometimes not (as with 2-way(I/II/III)).

In contrast, the propertied 2-way(I/II) network has a lower MCN-Average and lower MCN-Max than in the non-propertied case. This network is unique in that it converges to a fairly orderly topology, which we can best describe as "search clusters." In this topology, clusters of nodes are connected by search links, while nodes in different clusters are connected by index links. The effect is that nodes only send search messages to nodes in their own cluster. High coverage is achieved because each cluster has indexes from many or most of the nodes outside the cluster. Figure 4 shows an example. Each node in cluster 1 has an index link to a node in cluster 2 or 3. Thus, cluster 2 nodes can search cluster 1 nodes without any searches going to cluster 1. Clusters 2 and 3 would also have outgoing index links, although we have omitted them from the figure for clarity.

To see why the 2-way(I/II) network converges to this topology with propertied **connect()**s, recall that a type I link is a pair of search links (in opposite directions), while a type II link is a pair of index links (in opposite directions). Whenever a type II pair of index links is formed between two nodes, there is effectively a "boundary" between the two nodes, since there cannot be search links or a search path between those two nodes without forming a one-index cycle. As a result, the two nodes are necessarily in separate clusters. Similarly, if a type I pair of search links is formed between two nodes, those nodes are necessarily in the same cluster, since index links between the two nodes would also form a one-index-cycle. The result is that the ad hoc network self-organizes into an orderly topology. Note that propertied 2-way(I/II) networks still converge to search clusters even if we do not check for search-forks. As long as all links formed are type I or type II, every search-fork that occurred would also be a one-index-cycle. This fact makes it easier to deploy

a network using 2-way(I/II) connects, since only one property would have to be detected.

The search clusters topology is more efficient than the non-propertied 2-way(I/II) network because the nodes must only handle searches that originate within their own cluster. This is especially helpful in the case where $L_S >> L_U$, since much of the load in the network is search load and reducing the search load on a node is the key to efficiency.

The results for the load patterns $L_S = L_U$ and $L_S << L_U$ are similar: checking properties is beneficial for some networks, partially beneficial for others (e.g. by decreasing MCN-Average at the expense of higher MCN-Max), and detrimental for still others. For the 2-way(I/II) type specifically, checking properties is beneficial in the $L_S = L_U$ case but not in the $L_S << L_U$ scenario. This is because the efficiency of search clusters is due to extensive inter-cluster indexing, and when the update rate is high, this extensive indexing causes excessive load. Nonetheless, the search clusters topology that results from the ad hoc 2-way(I/II) **connect()** is an interesting example of how in some scenarios checking properties of search networks can lead to significantly more efficient networks.

Now that we have studied how to use **connect()** to build networks, we can compare the resulting ad hoc networks to supernode networks. For our comparison, we chose the "most-efficient" propertied and non-propertied ad hoc networks, that is, the networks with either the lowest MCN-Max or the lowest MCN-Average:

—*One-way*: The 1-way(f1.0/s0.5) network, in both the propertied and non-propertied forms.

—*Clusters*: The propertied 2-way(I/II) network, which forms "clusters" as discussed above.

—*Gnutella+IV*: The non-propertied 2-way(I/IV) network, which is essentially Gnutella augmented with type IV (search/index) links.

4.2.1 *Comparison to supernode networks.* We can compare the best ad hoc networks constructed using **connect()** to supernode networks. First, we must decide how many supernodes there will be in the network. We assume that supernodes are selected randomly, and we define P_{sn}, $0 \leq P_{sn} \leq 1$, as the probability that a node was chosen as a supernode. Then, in a network with N nodes, the expected number of supernodes is $P_{sn} \times N$. When $P_{sn} = 0$, one node was chosen as the supernode, since the network cannot function without at least one supernode. When $P_{sn} = 1$ all nodes are supernodes, which is equivalent to a pure search network like Gnutella.

To find a good value of P_{sn}, we ran an experiment where we varied P_{sn} from 0 to 1. The results for $L_S >> L_U$ are shown in Figure 5. As the figure shows, increasing the number of supernodes increases the MCN-Average. Supernodes are heavily loaded, and increasing the number of supernodes means that more nodes in the network are heavily loaded; thus, MCN-Average increases. At the same time, as the number of supernodes increases, MCN-Max initially decreases, and then levels off at about 91 messages per covered node. The decrease is due to the sharing of update load, while the leveling off is due to the search load. In a network with N nodes and $P_{sn} \times N$ supernodes, each supernode is responsible for $1/P_{sn}$

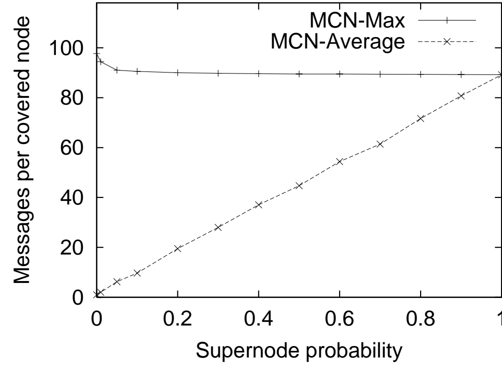


Fig. 5. Tuning supernode networks.

nodes (including itself) and has an indexing load of L_U/P_{sn}. Thus, increasing the number of supernodes decreases the update load on each supernode, decreasing the MCN-Max. However, each supernode is also responsible for $N \times L_S$ search load, since every search message is eventually forwarded to all supernodes. As a result, the search load is not affected by increasing the number of supernodes, and supernodes always have at least L_S messages per covered node (in Figure 5, 91 messages per covered node) regardless of the number of supernodes. The results for the load patterns $L_S = L_U$ and $L_S << L_U$ are similar, though the value at which MCN-Max levels off changes.

We chose three points in Figure 5 for our comparison:

—*Central indexing*: $P_{sn} = 0$

—*Part-supernodes*: $P_{sn} = 0.1$

—*Gnutella*: $P_{sn} = 1$

Besides the extremes of central indexing and Gnutella, part-supernodes is interesting because $P_{sn} = 0.1$ is the smallest P_{sn} after the "knee" in Figure 5.

Next, we can compare the selected supernode networks to networks constructed using our ad hoc techniques. Figure 6 shows the comparison for $L_S >> L_U$. Note that MCN-Average for central-indexing is 0.993, not zero. Three of the ad hoc networks (Gnutella+IV and the one-way networks) have equal or higher MCN-Max than the supernode networks. One network (propertied one-way) has a low MCN-Average compared to Gnutella and part-supernodes; however, the propertied one-way network has a higher MCN-Average than the central-index supernode network. Clearly, those three ad hoc networks do not have a compelling performance advantage over supernode networks.

In contrast, the clusters ad hoc network does have a performance advantage because it has a significantly lower MCN-Max than any of the supernode networks. The minimum MCN-Max of any supernode network in the $L_S >> L_U$ scenario would be 91 messages per covered node (due to the search load as described above), and all the supernode networks in Figure 6 have a MCN-Max of at least 91. In comparison, the clusters network has an MCN-Max of only 30.1 messages per covered

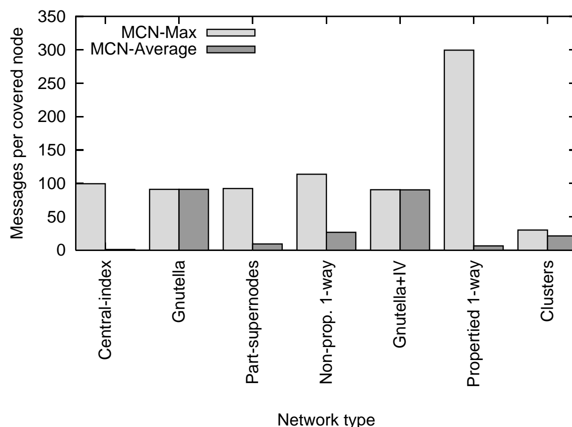


Fig. 6. Ad hoc **connect()** networks compared to existing networks.

node. This indicates that a clusters network is more appropriate than a supernode network for situations where there are few or no nodes capable of handling the high load of supernodes. Clusters does have a higher MCN-Average than supernode networks, which indicates that the decrease in MCN-Max is achieved by spreading load more evenly to all nodes. This load sharing is achieved despite the fact that nodes were not aiming for a specific topology but were instead using the ad hoc **connect()** operation to self-organize.

As we change the load patterns, the advantage of cluster networks decreases, while other ad hoc networks remain less efficient than supernode networks. When $L_S = L_U$, cluster networks have a roughly equal MCN-Max as Gnutella networks, and a lower MCN-Max than the other supernode types. When $L_S << L_U$, supernode networks are clearly better. This relationship can be seen more clearly in Figure 7, which shows cluster networks and part-supernode networks for various load patterns. On the left of the figure, where updates outnumber searches, part-supernodes are superior, with a lower MCN-Max and MCN-Average than cluster networks. As the proportion of searches grows, part-supernode networks become less efficient, while the efficiency of cluster networks improves. When there are at least as many searches as updates, the MCN-Max of cluster networks is less than MCN-Max of part-supernode networks, while the MCN-Average of cluster networks is within a factor of 2.7 or less of the MCN-Average of part-supernode networks.

Our results show that cluster networks achieve better load balancing than supernode networks when $L_S > L_U$. We expect this load scenario to be an important and common case. For multimedia filesharing, searches are likely to outnumber updates as nodes often submit multiple queries before finding an object of interest and adding it to their shared repository. Other applications are also likely to have $L_S > L_U$. For example, in a peer-to-peer digital library, users are likely to search for and download digital objects for their personal use without necessarily adding them to the collection at their local library and causing that library to issue an index update message. For these applications, there is a compelling case for using

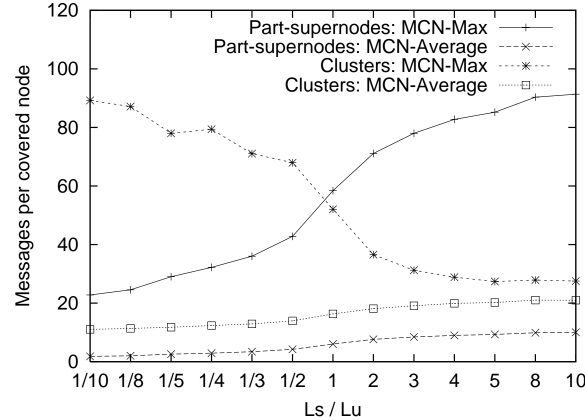


Fig. 7. Cluster networks versus supernode networks for various load patterns.

the clusters network instead of a supernode network, as clusters most effectively utilizes the network's resources to satisfy queries.

There are two other methods used in existing systems to tune supernode networks. One method is to partition the network into subnetworks. Then, supernodes would only handle the search and index load from nodes in their subnetwork, and nodes would not be able to search nodes in other subnetworks. While this reduces the absolute load on supernodes, our simulations show that the MCN is unaffected, since the decrease in load is accompanied by a corresponding decrease in coverage. The second method is to place a time-to-live (TTL) value on search messages, so that messages are only forwarded a certain number of hops. This again reduces load, since fewer search messages reach each supernode. In our experiments, adding a TTL value decreased the MCN-Max somewhat for supernode networks. However, we observed a similar decrease in MCN-Max when we used TTL in the ad hoc networks. The relative performance between supernode networks and ad hoc networks remained unchanged.

4.3 The **break()** operation

Next, we studied the effects of the **break()** operation. To do this, we first attempted to find the break method and parameters that produced the most efficient network for each **connect()** type. Recall from Section 3.3 that there are four break methods (MostLoadedLink, MostLoadedLinks, MostLoadedType, MostLoadedLinkOfType), and two parameters: the break threshold B_T, and B_I, the interval between breaks. For each **connect()** network type, we tried different values of B_T and B_I with each of the break methods. In all, we simulated over three thousand combinations of network type, break method, B_T and B_I. An example of our results is shown in Figure 8. This figure shows the MCN of networks built with a 2-way(III) propertied **connect()** operation, and the MostLoadedLink break type. On the horizontal axis is the break threshold B_T, and on the vertical axis is messages per covered node. Several series are shown: MCN-Average and MCN-Max for several settings of the break interval. For example, “Interval=150” indicate

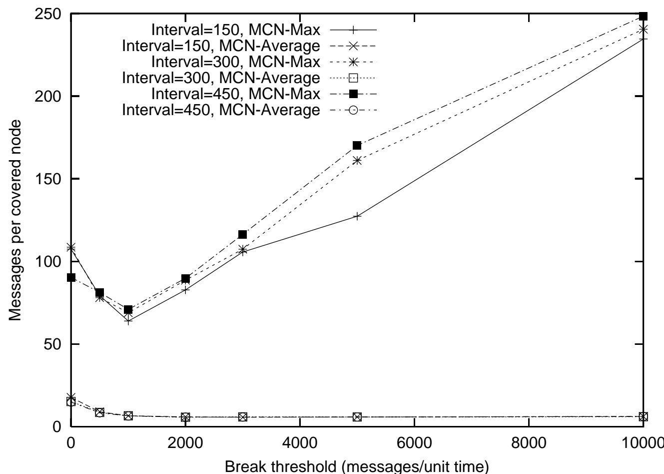


Fig. 8. Effect of break threshold with propertied 2-way(III) network and MostLoadedLink break type.

that a break occurred every 150 simulation time units. (We tried a total of six different intervals, but some are omitted to simplify the figure somewhat.) To place $B_I = 150$ time units in perspective, in our simulation a new node joined on average every 10 time units.

As the figure shows, the minimum MCN-Max occurred for all break intervals at a break threshold of 1000, while the MCN-Average decreases until a break threshold of 1000 and then remains flat. Using a similar analysis, we were able to find the break method, B_T and B_I that resulted in the minimum MCN-Max and MCN-Average for each of the **connect()** types. Our results indicate that a periodic MostLoadedLink or MostLoadedLinks break with a medium to high threshold is best. By performing the break periodically, the disruption to the coverage of the network is minimized. By avoiding an extremely high threshold, nodes ensure that some links are broken, but without the unnecessary disruption caused by a low threshold. MostLoadedType breaks too many links (for example, all search links) while MostLoadedLinkOfType may break a lightly loaded link of one type when a heavily loaded link of the other is actually the one causing problems.

We then examined whether **break()** and **connect()** together produced more efficient networks than **connect()** alone. We examined both propertied and non-propertied **connect()** operations. Some representative results are shown in Figure 9 for the case $L_S >> L_U$. First, the non-propertied 1-way network did not benefit at all from the **break()** operation. This is typical of many of the non-propertied networks. Even though overloaded nodes break links, these nodes quickly become overloaded again as new links are made in a random, undisciplined way.

One type of network that did benefit from **break()** is the non-propertied 2-way(I/II) network; using **break()** resulted in a lower MCN-Max and MCN-Average. Although broken links are still replaced randomly, over time the number of search links in the network decreases, and the proportion of index links increases, and the

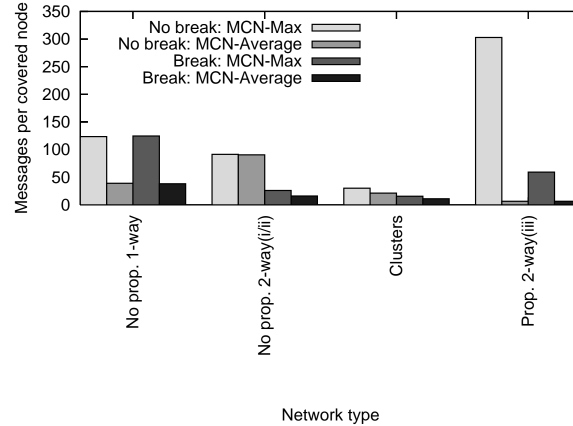


Fig. 9. The **break()** operation versus no **break()**.

network self-tunes for the high search load that is present. A 2-way(I/II) network is best able to effect this replacement of search links by index links because when a link is made, it is either pure search (type I) or pure index (type II).

Also, the propertied networks often benefit from the **break()** operation. In these networks, using **break()** usually decreased MCN-Max over the non-**break()** case. The decrease in MCN-Max was often accompanied by an increase or no change in MCN-Average. One example is the propertied 2-way(III) network shown in Figure 9. In this case, MCN-Max decreased by 80.4 percent when **break()** was used, while MCN-Average increased by six-tenths of one percent. In propertied networks, broken links are replaced in a more disciplined way by avoiding one-index-cycles or search-forks, and loads are able to effectively shed load. Sometimes this means that other nodes must become more loaded to take up the slack (and thus MCN-Average increases). In the case of the 2-way(III) network specifically, the **break()** operation reduces the occurrence of terminus nodes (see Section 4.2). Thus, a severe source of inefficiency is eliminated, and MCN-Max decreases without significantly increasing MCN-Average.

Another interesting case is the clusters network. In this case, using the **break()** operation is significantly beneficial, resulting in a 48.0 percent decrease in both MCN-Max and MCN-Average. The **break()** operation allows large clusters to break up into smaller clusters. This usually occurs through a process of nodes "seceding" from a large cluster one at a time and reforming into smaller clusters. Since nodes only receive search messages from other nodes in the same cluster, smaller clusters means that nodes receive fewer search messages, and this situation is better for the high search load scenario of $L_S >> L_U$. The search clusters of are also able to tune themselves for other load patterns. For example, when update messages are more prevalent than search messages, **break()** allows two clusters to join by replacing index links with search links, and thus nodes receive fewer update messages.

For other network types, the results for other load patterns ($L_S = L_U$ and

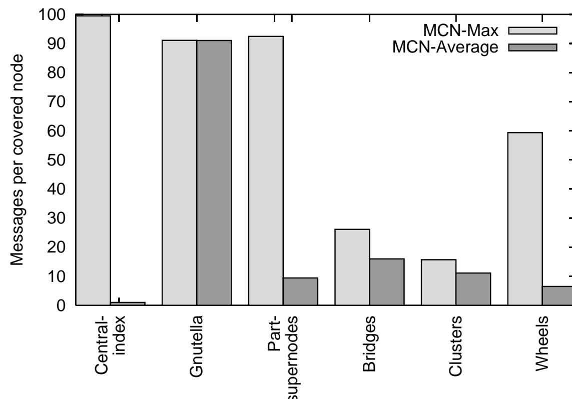


Fig. 10. Ad hoc **connect()** and **break()** networks compared to supernode networks.

$L_S << L_U$) are similar to the results in the $L_S >> L_U$ case: **break()** allows some networks (such as those discussed above) to tune themselves for higher efficiency for the current load pattern. Overall then, **break()** is an effective operation for network self-tuning.

4.3.1 *Comparison to supernode networks.* Now we can compare ad hoc networks with **connect()** and **break()** to supernode networks. We chose the following three networks that performed best under **break()**, achieving an MCN-Max less than 100:

—*Wheels*: This is a propertied 2-way(III) network. The resulting network tends to have multiple terminus nodes that are each the hub of a "wheel". The analogy is not precise, as the wheels network is more messy than the orderly clusters network.

—*Bridges*: This is a non-propertied 2-way(I/II) network. Without property checking, the network tends to form into disorderly clusters, where nodes in the same cluster have index link "bridges" between them.

—*Clusters*: As defined earlier, a propertied 2-way(I/II) network.

Figure 10 shows the comparison between **connect()/break()** ad hoc networks and supernode networks for the $L_S >> L_U$ scenario. As this figure shows, the ad hoc networks using **break()** achieve a smaller MCN-Max than any of the supernode networks. In the case of bridge and cluster networks, the MCN-Max is significantly lower; the bridge network has less than one third the MCN-Max of the supernode networks and the clusters network has about one sixth the MCN-Max of the supernode networks. This reduction in MCN-Max is achieved for the ad hoc network without unduly burdening the average nodes. The MCN-Average of the bridge network is only 1.7 times the MCN-Average of the part-supernode network, and the MCN-Average of the clusters network was only 1.2 times the part-supernode MCN-Average.

Our results indicate that our techniques compare favorably with the common case for supernode networks (e.g. some nodes are supernodes but not all) by spreading the load equally to all nodes. Moreover, in the case of a network where all nodes have roughly equivalent capacities, our techniques would fare better than any supernode network, since the low MCN-Max indicates that none of the nodes are overloaded. These results hold for the other load patterns ($L_S = L_U$ and $L_S << L_U$) as well: **break()** allows the bridge and cluster networks to tune themselves and spread load around more effectively than supernode networks.

Even if the network has nodes of widely varying capacities, our techniques perform quite well. Consider the wheels network, which has both a lower MCN-Max (by 35.8 percent) and a lower MCN-Average (by 30.9 percent) than the supernode network with one tenth supernodes (for $L_S >> L_U$). This means that in any scenario where part-supernodes is appropriate, the wheels network is more efficient for all nodes for that load pattern. For other load patterns ($L_S = L_U$ or $L_S << L_U$), the wheels network has a higher MCN-Average than supernode networks.

Finally, we ran an experiment where we used the **break()** operation in supernode networks. The **break()** operation executed in the same way as in the ad hoc networks, and broken links were replaced using the supernode network version of **connect()** (supernodes connected to each other while normal nodes connected to supernodes). Our results (not shown) indicate that **break()** is ineffective as a method for tuning supernode networks, as it did not significantly change MCN-Max or MCN-Average. The structure of the topology is too rigid; even when links are broken the same inherent traffic patterns remain. Ad hoc networks, with their flexible topologies, are much better suited to tuning with **break()**.

We can summarize our results as follows:

—Constructing a network in an autonomous, ad hoc fashion does not hurt the efficiency of the network, even though there is no classification of nodes or rigid structure.

—On the contrary, such ad hoc techniques can lead to a very efficient network, a network that is better than a supernode network for both the situation where some nodes are more powerful than others, and also for the scenario where nodes have roughly equivalent capabilities.

—The **break()** operation is an effective way to reduce load on all nodes in an ad hoc network.

—Checking for properties when performing **connect()** is always beneficial for the 2-way(I/II) network (clusters), and beneficial for some other types of networks when the **break()** operation is used.

—The clusters network (with or without **break()**s) and the bridge network (with **break()**s) are more effective than supernode networks at spreading the load to all nodes without overloading nodes. This is important for search-heavy applications where few or none of the nodes have enough capacity to act as supernodes.

—The wheels network (with **break()**s) is more effective than supernode networks at reducing load on all nodes, and is useful for situations where some nodes have higher capacity than others.

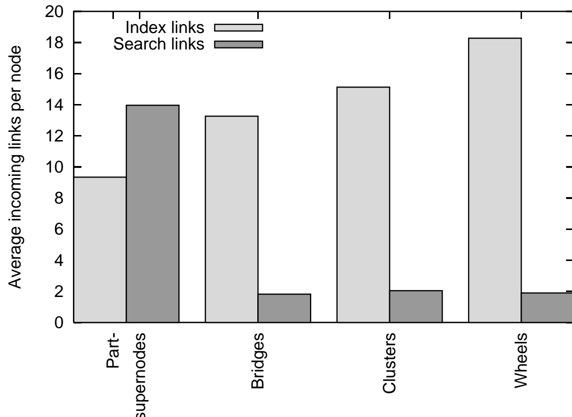


Fig. 11. Average incoming links per node.

4.4 Index overhead

One primary reason that networks constructed using our techniques can outperform supernode networks is that our techniques allow nodes to strategically and adaptively place indexes in the network to reduce overall load. However, this means that our ad hoc networks may create a large number of indexes, trading off index storage space to achieve overall load efficiency. In this section we examine the overhead on nodes resulting from storing indexes.

Our simulation does not explicitly model index sizes because index size depends on the type of query that needs to be supported. For example, if queries only search over titles or other short metadata, then the indexes can be very small compared to the size of the content being indexed. On the other hand, if queries search over the full content (e.g., full text searches), then the index size can be roughly the same size as the content (e.g., full inverted index). Of course, indexes can be compressed in various ways, so the size requirements can be reduced, depending on the search and update overhead we want to incur.

Even though we do not model index sizes explicitly, we can count the number of incoming index links for each node to get an idea of how many indexes that node has to store. We counted incoming index links for bridge, cluster, wheel and supernode networks at the same time as the load measurements of Figure 10 were taken (e.g., at the network steady state). We also counted, for comparison, the number of incoming search links.

The results for the $L_S >> L_U$ scenario are shown in Figure 11. (Our results for other load scenarios are similar.) Note that the values for "part-supernodes" represent incoming links only for supernodes, since normal nodes do not have any incoming links in a supernode network. As the figure shows, our ad hoc networks have more incoming search links per node than supernode networks, by a factor of 1.4 for bridge networks, 1.6 for cluster networks and 2.0 for wheels networks. This result illustrates that our ad hoc networks must create many indexes to achieve the performance gains shown in Figure 10. However, even in the most index heavy

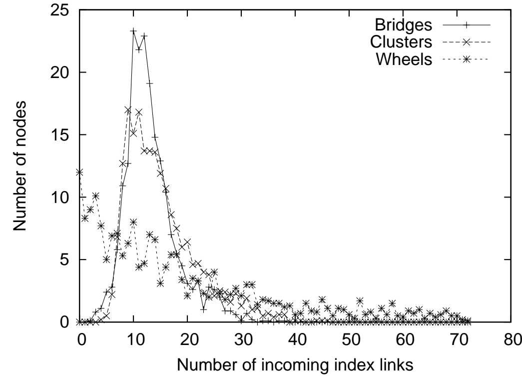


Fig. 12. Distribution of incoming index links.

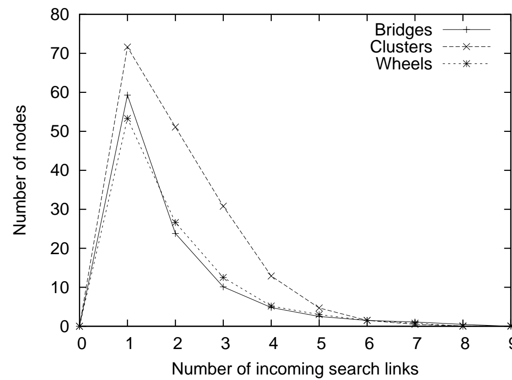


Fig. 13. Distribution of incoming search links.

network (wheels), nodes index only a fraction of the total content in the system (with an average of 18.3 incoming index links compared to 200 total nodes in the network). Ad hoc networks may not be appropriate in situations where indexes are very large or difficult to maintain (for example, the full-text indexes mentioned earlier) because of the extra indexes that are created compared to supernode networks. However, if individual indexes are small, or can be made small through compression or summarization techniques, then the performance gains of ad hoc networks can justify the extra indexing cost.

We also examined the distribution of the indexing load in our ad hoc networks. Figure 12 shows the distribution of incoming index links. This figure is a histogram, with the horizontal axis representing the number of incoming index links and the vertical axis representing the number of nodes that have that many incoming index links. (For comparison, Figure 13 shows the distribution of incoming search links.) As the figure shows, in the bridges and clusters networks, most nodes have roughly the average number of incoming index links, so that the storage costs for all nodes

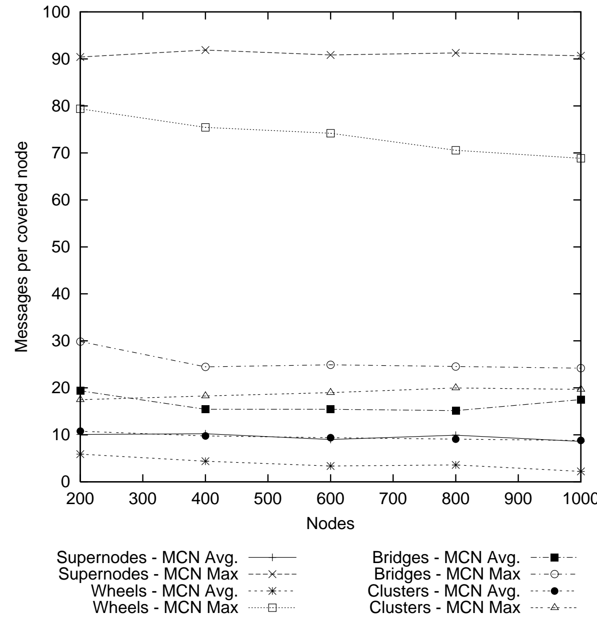


Fig. 14. Network size.

is roughly balanced. In contrast, in the wheels network, most nodes have a few incoming index links while a few nodes have many incoming index links. This fits our intuition from the previous section, that bridge and cluster networks are best at equalizing the load across all nodes, while wheels networks are most appropriate for more heterogeneous networks where nodes have varying capabilities.

4.5 Scale

The above results are for networks with 200 nodes. We also ran experiments in which we varied the number of nodes between 200 and 1000. The results give us an idea of the efficiency trends in the network as we increase the number of nodes. Since it is infeasible to perform experiments with very large networks (e.g., millions of nodes), we can estimate how efficient large ad hoc networks are likely to be.

Figure 14 shows the results. The figure shows that both MCN-Max and MCN-Average remains relatively flat for all four network types as the number of nodes increases. In particular, ad hoc networks continue to exhibit the efficiency benefits compared to supernode networks that were discussed in Sections 4.2 and 4.3. In some cases (such as the wheels network) the efficiency of the ad hoc network actually improves as more nodes join the network, since a larger network has more nodes that can share load. Although a larger network also means that more nodes are generating load, the ad hoc networks effectively use the extra resources so that the overall effect is a more efficient network.

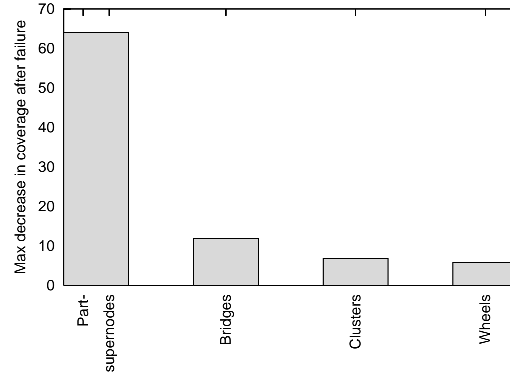


Fig. 15. Fault susceptibility.

4.6 Other metrics

So far we have dealt with the load in the network, since a primary limitation of peer-to-peer systems is scalability limitations due to node overloading. However, there are other reasons why a given topology may be preferred. We also examined three other metrics: the fault tolerance, search latency and coverage of a network.

First, we examined fault tolerance. Peer-to-peer networks can become partially or fully partitioned after a node fails. To quantify this effect, we measured *fault susceptibility*: the maximum decrease in coverage that can be caused by a node failure. The results are shown in Figure 15. The fault susceptibility metric is calculated at the same time the MCN metric was; that is, after the Nth node had connected. As the figure shows, a failure in a supernode network can cause a large decrease in coverage. This is because if a supernode fails, all of its connected normal nodes become completely disconnected, and until they reconnect to the network they cannot search or be searched. In contrast, the susceptibility to faults is lower in bridges, clusters and wheels networks. This is because these ad hoc networks share load more evenly than in a supernode network, and thus no single node is vitally important. A failure in one of these ad hoc networks causes some nodes to lose coverage, as the failed node is not searchable and is no longer providing indexing services, but most nodes are unaffected.

Next, we looked at the time it takes for searches to complete. If search messages must travel many hops before the node that generated the search gets all of its results, then the search will take a long time. We define *search latency* for a node as the longest search path between that node and any other node, and take the average node search latency to get the search latency for the network. Figure 16 shows the results. (Again, search latency is calculated after the Nth node connects.) The figure shows that the wheels network, with an average of 2.1 hops, has a better search latency than the supernode network, at 3.6 hops on average. The clusters network has a longer search latency than supernodes, but only by ten percent. The longest search latency is for the bridges network (at 4.8 hops on average) but even this is not terrible compared to supernode networks. In short, ad hoc networks compare favorably with supernodes in terms of search latency.

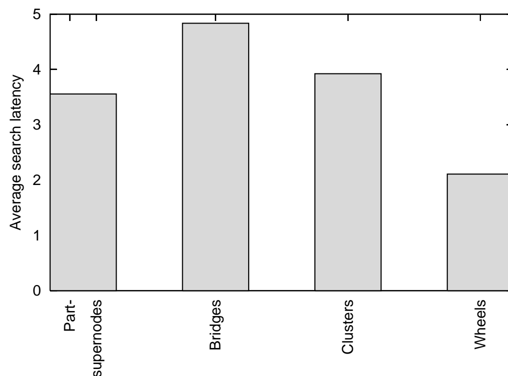


Fig. 16. Search latency.

Finally, we examined coverage. It is clearly overkill for an information discovery network to require 100 percent coverage, since in a large network useful documents can be found by examining only a portion of the network, and users do not require perfect recall. This is the case in the most popular existing networks such as Gnutella and Kazaa, where only a portion of the network is searched. However, it is still desirable to have a reasonable amount of coverage, to ensure that documents can be found. This reason is why we stipulated that our ad hoc networks must have relatively high coverage on average (e.g., at least forty percent of the network). Sometimes it was possible to achieve better MCN-Max or MCN-Average in lower coverage networks, and the network designer must decide what the appropriate coverage level is. Our results (not shown) indicate that the coverage in ad hoc networks we studied, while adequate, was not exceptionally high, with between 44 and 56 percent of the coverage of supernode networks. Clearly, an important area for continued work is to find ways to improve coverage without sacrificing efficiency in terms of MCN, if high coverage is important for a given application. One possibility is for nodes to connect to nodes that provide the largest increase in coverage, rather than selecting nodes randomly. Another possibility is to break links that have a low coverage to load ratio.

We also examined the distribution of coverage in the network. These results (not shown) indicate that for the wheels network, most nodes had coverage roughly equal to the average coverage in the network. However, in the clusters network, most nodes had relatively high coverage, reaching 65 percent of the network or more, while about one quarter of the nodes had low coverage, at 40 percent or less. In the bridges network, there was a larger proportion (roughly half) of nodes with low coverage. Another important area for future work is finding ways to improve coverage for all nodes in the clusters and bridges networks without sacrificing efficiency.

5. RELATED WORK

Several studies [Saroiu et al. 2002; Ripeanu and Foster 2002] have focused on characterizing networks such as Gnutella, and many researchers agree such networks

cannot easily scale. Pandurangan et al have looked at techniques for constructing Gnutella-like networks with desirable properties [Pandurangan et al. 2001], although their focus is on pure-search networks without index links. Other flooding-like networks that employ indexing similar to our index links include routing index networks [Crespo and Garcia-Molina 2002], local indexing networks [Yang and Garcia-Molina 2002], result caching networks [Bhattacharjee 2003] and of course supernode networks [kaz 2003; Yang and Garcia-Molina 2003; Nejdl et al. 2003]. These systems are specific instances of using indexing in peer-to-peer systems, while we are attempting to construct a general framework. Moreover, our focus is on ad hoc networks that can dynamically adapt to changing conditions, rather that being bound to a specific structure as in many of these systems.

Some researchers have abandonded flooding networks in favor of highly structured distributed hash tables (DHTs), such as Chord [Stoica et al. 2001] and CAN [Ratnasamy et al. 2001], in an attempt to provide a much more scalable lookup service. DHTs focus on finding the location of an object whose name is known, but often rely on a separate mechanism for information discovery (as pointed out in [Ratnasamy et al. 2001]). The search networks we study here combine the operations of information discovery and object location. Moreover, DHTs are not yet widely deployed, and the popularity of Gnutella and Kazaa suggest that flooding networks are still worthy of study despite the advent of DHTs.

Still others have proposed improvements to unstructured networks, such as using random walk searches [Lv et al. 2002]. Content is proactively replicated to improve the recall of the random walk [Cohen and Shenker 2002]. However, content is replicated to random nodes, while our techniques allow replication of indexes adaptively to place them at points in the network where they do the most to improve efficiency. Moreover, random walks may travel many nodes before finding content, which increases the latency of searches. However, random walking may be useful in the networks we study here, and further study is required. Another proposed alternative is to use information from past queries to aid routing in unstructured networks [Kalogeraki et al. 2002]. Again, it may be useful to augment our ad hoc networks with such a routing strategy.

Lv, Ratnasamy and Shenker [2002] have proposed a mechanism for topology adaptation that is similar in spirit to our approach. However, our techniques differ from their work in several ways. First, we treat index links as first class objects, to allow adaptive and strategic placement of indexes to improve scalability. While [Lv et al. 2002] mentions proactive content replication as a possible extension to their system, explicit replication of indexes or content is not examined. Second, the system in [Lv et al. 2002] uses random walks, where a node forwards a query to a single random neighbor to produce depth-first search of the network. Queries may have to walk for many hops before finding content. Our system assumes that nodes forward queries on all outgoing search links, producing breadth-first search instead. Traditionally, the breadth-first approach used in our system has much better response time (because messages are forwarded in parallel) but much worse scalability than random walk approaches (since more messages are propagated). Our system retains the response time of breadth-first approaches but improves overall network scalability through index replication. Thus, our system aims to

provide both good response time and scalability, unlike random walks which only provide scalability. Third, the techniques in [Lv et al. 2002] require nodes to track the capacities of neighbors, necessitating extra control traffic and state to keep capacity information updated. In contrast, in our system, nodes are only responsible for tracking their own capacity. If they connect to an overloaded node, this node can break the link, resulting in effective load balancing without the need to explicitly measure the capacity of neighboring nodes. It may be possible that having capacity information can help guide nodes in our system in making connections, and we can extend our techniques with such a mechanism if the cost of gathering such information is justified by increased efficiency. We have not yet examined this possibility in detail.

Several investigators have to reorganize search networks using content semantics or topics [Khambatti et al. 2003; Loeser et al. 2003; Tang et al. 2003; Bawa et al. 2003]. It may be possible to extend our techniques to take semantics into consideration, for example by preferentially **connect()**-ing to sites with similar content, although we have not yet looked into this.

Self-supervising networks are similar to Dijkstra's [1974] concept of a self-stabilizing system in which the system stabilizes through the actions of individual nodes. Self-supervising work in peer-to-peer includes Anthill [Babaoglu et al. 2002], a framework for developing self-organizing and self-tuning networks. The techniques in Anthill (such as genetic programming) are more complex than the simple operations we examine here.

Peer-to-peer search is so far most popular in filesharing applications, but several other applications have been proposed. These include data storage for ubiquitous computing [Kubiatowicz et al. 2000], privacy-preserving publishing [Dingledine et al. 2000], and content discovery in distributed digital archives [Cooper and Garcia-Molina 2002]. Our techniques can be used to provide efficient and adaptive information discovery services for these applications. Others have proposed models and architectures for peer-to-peer database systems [Huebsch et al. 2003; Halevy et al. 2003; Bernstein et al. 2002; Kementsietsidis et al. 2003; Gribble et al. 2001]. These systems focus on structured query processing, which has different challenges and requires different techniques than the information discovery and retrieval problem we focus on here.

6. CONCLUSION

Peer-to-peer search networks are a popular and effective architecture for information discovery in massively distributed digital repositories. Many researchers have suggested that flooding-based search networks should be replaced by other search techniques. However, we believe that the flooding model is still extremely interesting and viable because of its simplicity, flexibility and robustness. We have investigated constructing flooding-based search networks that are built in an ad hoc manner, without restricting *a priori* which nodes can connect or what types of information they can exchange. In order to make these ad hoc networks efficient, we have made them self-supervising, so that the network topology adapts to be as efficient as possible.

Our techniques improve over existing networks, such as supernode networks, in

several key situations. If there are no nodes in the network able to take on the overwhelming burden of becoming a supernode, our ad hoc networks can be tuned to more evenly distribute load in the network. This is done by dropping links using the **break()** operation to shed load, by checking to ensure that efficiency properties are preserved when **connect()**ing to the network, or by doing both. Even if there are nodes that have higher capacity, our techniques allow an ad hoc network to organize and tune itself into a structure that is more efficient than existing supernode networks for both the supernode and the normal node. Our results indicate that ad hoc, self-supervising networks are an effective architecture for peer-to-peer search.

It may be possible to extend our techniques in several ways. For example, we have focused on load efficiency, but in some applications other metrics such as fault tolerance or response time might be more important. Nodes could still use **connect()** and **break()** to tune for these other metrics. Thus, a node might break an outgoing search link if searches sent along that link have a high response time. Moreover, there may be other properties that it would be useful to detect during a propertied **connect()**, perhaps utilizing results from the graph theory literature, either to tune for load efficiency or to tune for these other metrics. Finally, we have assumed a general peer-to-peer search application, and focused on general message flow properties in our evaluation. The details of certain applications might not match the assumptions we have made, and therefore it would be useful to construct workloads that are specifically tuned for those applications. These extensions to the techniques being presented here are being considered as part of our ongoing work.

REFERENCES

2003. Gnutella. http://gnutella.wego.com.

2003. Kazaa. http://www.kazaa.com.

Babaoglu, O., Meling, H., and Montresor, A. 2002. Anthill: A framework for the developement of agent-based peer-to-peer systems. In *Proc. of Int'l Conf. on Distributed Computing Systems (ICDCS)*.

Bawa, M., Jr., R. J. B., Rajagopalan, S., and Shekita, E. 2003. Make it fresh, make it quick — searching a network of personal webservers. In *Proc. Int'l World Wide Web Conf.*

Bernstein, P., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. 2002. Data management for peer-to-peer computing: A vision. In *Proc. WebDB Workshop*.

Bhattacharjee, B. 2003. Efficient peer-to-peer searches using result-caching. In *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*.

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

Cooper, B. F. and Garcia-Molina, H. 2002. Peer-to-peer data trading to preserve information. *ACM Transactions on Information Systems 20,* 2 (Apr.), 133–170.

Cooper, B. F. and Garcia-Molina, H. 2003a. SIL: Modeling and measuring scalable peer-to-peer search networks. In *Proc. of the Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing*.

Cooper, B. F. and Garcia-Molina, H. 2003b. SIL: Modeling and measuring scalable peer-to-peer search networks. Technical Report, available at http://www.cc.gatech.edu/ cooperb/-pubs/searchnetsext.pdf.

Crespo, A. and Garcia-Molina, H. 2002. Routing indices for peer-to-peer systems. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*.

DIJKSTRA, E. 1974. Self stabilizing systems in spite of distributed control. *Communications of the ACM 17,* 11 (Nov.), 643–644.

DINGLEDINE, R., FREEDMAN, M., AND MOLNAR, D. 2000. The FreeHaven Project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability.*

GRIBBLE, S., HALEVY, A., IVES, Z., RODRIG, M., AND SUCIU, D. 2001. What can databases do for peer-to-peer. In *Proc. WebDB Workshop.*

HALEVY, A., IVES, Z., MORK, P., AND TATARINOV, I. 2003. Piazza: Data management infrastructure for semantic web applications. In *Proc. Int'l World Wide Web Conf.*

HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., AND SHENKER, S. 2003. Querying the Internet with PIER. In *Proc. Int'l Conf. on Very Large Databases (VLDB).*

KALOGERAKI, V., GUNOPULOS, D., AND ZEINALIPOUR-YAZTI, D. 2002. A local search mechanism for peer-to-peer networks. In *Proc. Conf. on Information and Knowledge Management (CIKM).*

KEMENTSIETSIDIS, A., ARENAS, M., AND MILLER, R. 2003. Mapping data in peer-to-peer systems: semantics and algorithmic issues. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data.*

KHAMBATTI, M., RYU, K., AND DASGUPTA, P. 2003. Structuring peer-to-peer networks using interest-based communities. In *Proc. Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing.*

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

LOESER, A., NAUMANN, F., SIBERSKI, W., NEJDL, W., AND THADEN, U. 2003. Semantic overlay clusters within peer-to-peer networks. In *Proc. Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing.*

LV, Q., CAO, P., COHEN, E., LI, K., AND SHENKER, S. 2002. Search and replication in unstructured peer-to-peer networks. In *Proc. of ACM Int'l Conf. on Supercomputing (ICS).*

LV, Q., RATNASAMY, S., AND SHENKER, S. 2002. Can heterogeneity make gnutella scalable? In *Proc. of the 1st Int'l Workshop on Peer to Peer Systems (IPTPS).*

NEJDL, W., WOLPERS, M., SIBERSKI, W., SCHMITZ, C., SCHLOSSER, M., BRUNKHORST, I., AND LOESER, A. 2003. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proc. WWW.*

PANDURANGAN, G., RAGHAVAN, P., AND UPFAL, E. 2001. Building low-diameter P2P networks. In *Proc. IEEE Symposium on Foundations of Computer Science (FOCS).*

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content-addressable network. In *Proc. SIGCOMM Conf.*

RIPEANU, M. AND FOSTER, I. 2002. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *Proc. of the 1st Int'l Workshop on Peer to Peer Systems (IPTPS).*

SAROIU, S., GUMMADI, K., AND GRIBBLE, S. 2002. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Conferencing and Networking.*

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM Conf.*

TANG, C., XU, Z., AND DWARKADAS, S. 2003. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. SIGCOMM Conf.*

YANG, B. AND GARCIA-MOLINA, H. 2002. Efficient search in peer-to-peer networks. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS).*

YANG, B. AND GARCIA-MOLINA, H. 2003. Designing a super-peer network. In *Proc. Int'l Conf. on Data Engineering (ICDE).*

YANG, B., VINOGRAD, P., AND GARCIA-MOLINA, H. 2004. Evaluating GUESS and non-forwarding peer-to-peer search. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS).*

Received Month Year; revised Month Year; accepted Month Year