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

Brian F. Cooper and Hector Garcia-Molina
Department of Computer Science
Stanford University
{cooperb,hector}@db.stanford.edu

Abstract

Peer-to-peer search networks are a popular and widely deployed means of searching massively distributed digital object repositories. Unfortunately, as such networks grow, they place an increasingly overwhelming load on some or all of the participating nodes. We examine how to reduce the load on nodes by allowing them to self-organize into a relatively efficient network, and then self-tune to make the network even more efficient. Unlike previously studied architectures, our "ad hoc, self-supervising" networks avoid restrictions on who a node can connect to or what information can be exchanged. This makes the network topology quite flexible and tuneable. Our results indicate that our ad hoc networks are more efficient than popular supernode topologies for several important scenarios.

1 Introduction

Peer-to-peer search networks are an effective mechanism for sharing information between large numbers of users. On a typical day, existing networks such as Kazaa support several 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 the fact that even though none of the network peers have to be more powerful than a desktop computer, the aggregate resources of millions of desktop computers can be harnessed to provide a very powerful search service.

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, using up resources that users may prefer to spend on other tasks. 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 (for example by throttling bandwidth) or may even convince users not to join networks at all.

In the past few years, a great deal of research effort has been focused on improving the efficiency and scalability of search networks, so that they can be used for a variety of tasks beyond the traditional role of multimedia file-sharing. Most of the proposed solutions have fixed rules about who nodes can connect to. For example, in a supernode network, peers are designated as "super-peers" or "normal peers", and normal peers are only allowed to connect to super-peers. In distributed hash tables such as Chord [16] or CAN [13], a node's neighbors are a fixed function of the node's id. Also, in many proposed solutions the types of connections between nodes are prescribed. For example, in supernode networks, super-peers index the content of normal nodes but do not send indexing information to other superpeers.

While each of these solutions have distinct advantages, we wanted to explore the possibility of a completely decentralized network of autonomous nodes, where networks are built in a more flexible and ad hoc way. In such ad hoc networks, nodes are not restricted to certain neighbors. Moreover, nodes can decide whether to send to their neighbors queries, index entries, or both, and similarly whether to accept queries, index entries or both from their neighbors. Such a network is likely to be inefficient unless it can become *self-supervising*; that is, the network must self-organize into a relatively efficient and effective topology, and must self-tune to improve efficiency and reduce load.

The goal of this paper is to examine such ad-hoc, self-supervising networks to see how they compare to more rigidly structured search networks. To do this, we have experimented in the context of a content discovery network like Gnutella [1, 14] or Kazaa [2] to see how nodes can make local decisions that are both beneficial to themselves and good for the network as a whole. Other investigators have also examined search in relatively unstructured networks, and our work differs from previous studies (such as [10]) in three key ways. First, in our ad hoc networks, nodes connect to each other using either "search links" or "index links." Index links replicate only an index over a node's data and not the data itself, allowing one node to search another's repository without having to actually send a query to the other node or store a copy of the other node's data. Our index links are similar to those in a supernode network, where supernodes store indexes of normal nodes, but without the fixed connection rules and rigid separation between "super-peers" and "normal peers." Second, the networks we study become self-supervising through the use of two very simple operations: connect(), where nodes form ad hoc connections to other nodes, and break(), where nodes that are overloaded simply break links to shed load. This allows a network to tune itself for efficiency in a simpler way than previous solutions such as [11], since we do not require overloaded nodes to track their neighbors' capacity or find other nodes to explicitly take on load. Third, ad hoc networks do not dictate which nodes are responsible for which content, as in distributed hash tables (where content is assigned to nodes based on a hash function [16]) or random walk networks (where proactive replication requires some nodes to store data even if they are not interested in it [10]).

Our results indicate that in several situations, such ad hoc local decision making results in networks that are very efficient, more efficient than popular existing solutions such as supernode networks. We have studied the actions available to nodes in the network, such as how one node connects to another. There are several interesting connection methods, some of which result in a "messy" but efficient network, and some of which converge to an orderly topology even though connections are made randomly and independantly. Moreover, we have been able to optimize ad hoc networks both for a heterogenous network that contains nodes of differing capabilities as well as for a more homogenous network where all nodes have roughly similar capabilities. Thus, ad hoc networks that are maintainted using connect() and break() can self-tune for the network's capabilities and load profile.

In this paper, we examine how an ad hoc, self-supervising peer-to-peer search network is built and maintained, and evaluate such networks in terms of efficiency. Specifically, we make the following contributions:

- We discuss how to build ad hoc networks from search links and index links. Such networks leverage the same basic search and indexing primitives of existing networks to achieve efficiency but do not restrict a node's neighbors or what information can be exchanged.
- We introduce the operations of connect() and break(), which are simple, local operations performed by peers. These operations are the key to introducing self-supervising properties into the network.
- 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.

This paper is organized is follows. In Section 2, we present our model of peer-to-peer search networks. Section 3 presents the basic techniques for dynamically constructing search networks in a self-supervising manner. In Section 4 we present 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

In a P2P search network, all of the nodes 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 finds an object matching the query, a result is returned to the user indicating the location of the object. The user can then directly contact the node holding the object and retrieve it.

Nodes in a search network connect to each other to form a partially-connected overlay over a fully connected network infrastructure such as the Internet. There are two types of connections in this overlay network:

- *Search links*, which are used to forward queries between nodes
- *Index links*, which are used to send copies of content indexes between nodes.

Index links reduce the need to flood the entire network with queries, since one node A can perform searching operations over another 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. For example, in order to aid in processing searches, each node might construct an inverted list of words in the content of its digital objects, a list of the object titles, or some other indexing structure. The node B sends its index to node A via an index link. 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.

We have developed a simple and useful model, the Search/Index Link (SIL) model, for studying and visualizing search networks. The SIL model represents a search

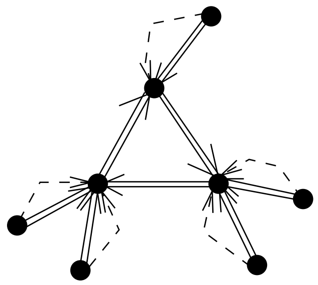


Figure 1: A supernode network represented as a SIL graph.

network as a directed graph, with vertices representing P2P nodes and different types of edges for the different types of connections between nodes in the overlay network. In the full SIL model, there are four types of edges:

- A *forwarding search link* (FSL) from vertex X to vertex Y ($X \Longrightarrow Y$) represents an overlay network link that carries search messages from peer X to peer Y. Peer Y processes the query and also forwards it on outgoing FSLs.
- A *non-forwarding search link* (NSL) $X \longrightarrow Y$ represents an overlay link that carries search messages from peer X to peer Y. Peer Y processes the query but does not forward it.
- A *forwarding index link* (FIL) $X \dashrightarrow Y$ represents an overlay link that carries index updates from peer X to peer Y. These index updates inform Y about new, modified or deleted content at peer X. Peer Y integrates the index updates into its own index, and also forwards the updates on outgoing FILs.
- A *non-forwarding index link* (NIL) $X \dashrightarrow Y$ represents an overlay link that carries index updates from X to Y. Peer Y should add the updates to its own index but does not forward them.

For example, a supernode network such as Kazaa can be represented using SIL, as shown in Figure 1. Supernode vertices are connected to one another using FSLs ($\Longrightarrow$), while vertices representing normal nodes are connected to supernodes using an FSL and a NIL ($\dashrightarrow$).

In this paper, we use the SIL model both as a visualization tool for drawing search networks and as a framework for analyzing the properties of networks that result from our techniques. By constructing SIL graphs and examining their graph structure, we can gain insights into the behavior of the P2P search networks represented by the SIL graphs.

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 wished 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.

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. 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.

After A has been in the network for a while, it 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 be receiving more search messages from a link $A \Longleftarrow F$ than it is receiving from any other link. Then, A should probably drop the link $A \Longleftarrow F$. If A is still overloaded, it can drop more links. Nodes that have been disconnected from A (such as F) can now perform connect() to replace the broken connection to A. For example, F may decide to form a search link to nodes other than A, or may decide to form an index link $F \dashleftarrow A$ so that F is still able to search A's content.

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. Moreover, the network can tune itself for the load pattern it is experiencing. For example, if nodes are overloaded with search messages, they will drop search links, which can then be replaced with index links, evolving the network into a more index-centric structure. Similarly, if nodes are overloaded with update messages, index links can be replaced by search links, shifting the network into a more search-centric mode.

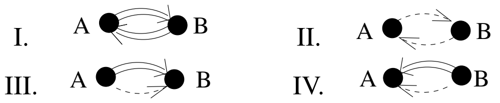
In this section, we examine the decisions available to nodes during connect() and break(). During connect(), a node must decide which types of links to form, while during break(), a node must decide which link or links to disconnect. We focus on graphs that are constructed from FSLs and NILs. This simplifies our analysis, since there are fewer graphs to consider than in the full SIL model. We have chosen FSLs and NILs because these edge types are sufficient to represent the most popular existing search network topologies, such as Gnutella or supernode networks such as Kazaa. For clarity, we will use the term "search links" to mean FSLs, and "index links" to mean NILs.

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.

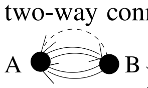
When performing a one-way connect(), a node must decide which link to form with a particular neighbor, and the easiest thing to do is to randomly choose the link type and direction. This process is described by two parameters: P_{sl}, the probability that a search link is chosen instead of an index link, 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.

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.

There are other ways of connecting besides one-way and two-way connect() (for example, a connect() that forms 

) 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.

The probability parameters P_{sl}, P_f, P_I, P_{II} and so on can be set in a number of ways. One way is for a node to adaptively learn good values, depending on network conditions. For example, a node using one-way connect() may become overloaded with search messages, and thus decrease P_{sl}. Another way to set the probability parameters is to run experiments to find values that tend to produce efficient networks in many cases, even though they may not be optimal for specific nodes. This latter approach is the one we take, and in Section 4, we examine the values of the parameters that lead to efficient networks (networks where nodes are comparatively lightly loaded.)

3.1.1 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 represented by the SIL graph 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

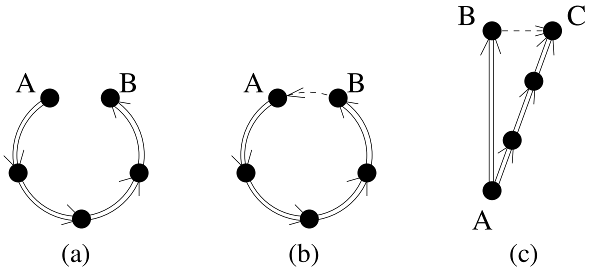


Figure 2: Networks: (a) search path, (b) one-index-cycle, (c) search-fork

index link does not benefit any node. We say that the link is *redundant*.

More formally, we define *redundancy* as a property of a graph where an edge can be removed without reducing the *coverage* for any vertex. The *coverage* of a vertex A is the number of other vertices that A can search. 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 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. 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, 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.

3.2 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

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	

Table 1: Tuning connect() and break().

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.

4 Evaluation

We have evaluated our techniques by simulating network construction and analyzing the properties of the resulting networks. 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 1. 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 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.

We use a load efficiency metric both for the ad hoc network tuning process and for comparing ad hoc networks to supernode networks. The specific metric we used in our evaluations is Messages per Covered Node (MCN), which is calculated as the load on 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. 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 half 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.

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 2; these parameters are discussed in detail below. The base values in Table 2 indicate the values used in the results we report. We tried a variety of val-

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

Table 2: Simulation parameters.

ues for these parameters, and the overall results remained consistent except where noted.

In our simulations, we model a peer-to-peer search network as a SIL graph, with vertices representing nodes and edges representing search and index links (as described in Section 2). The graph starts empty, and vertices are "born" at random intervals, on average every I_B time ticks. When a vertex is born, it selects random vertices already in the graph and connect()s to those vertices, until the new vertex has at least M incoming or outgoing edges. If the vertex is unable to form M edges, say because there are not enough vertices in the graph, then that vertex 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 $M = 20$, which represents between 5 to 20 neighbors for a node, since up to four links may connect with the same 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, edges to break, and second, the selected edges are removed from the graph. This represents a simplified version of the break() operation in a real network, where nodes would continuously 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 edges were removed from the graph. Lifetime values were chosen randomly from a normal distribution with a mean of μ_L and standard deviation σ_L. 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 graph 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 vertex 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 edge 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

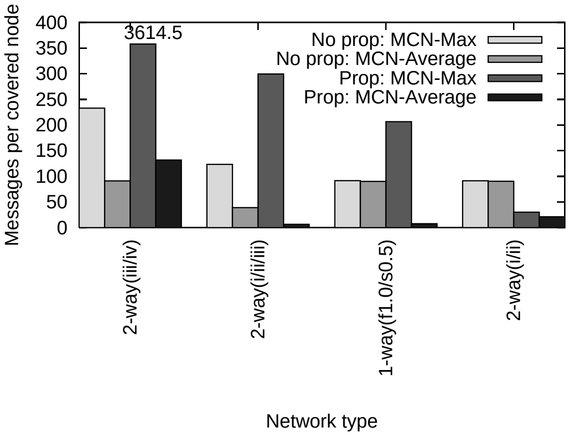
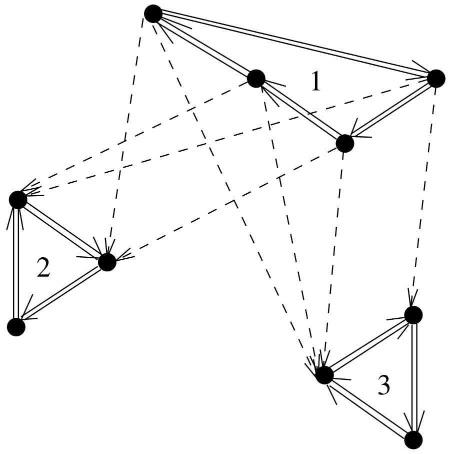


Figure 3: Non-propertied connect() versus propertied connect().



(Some index links have been omitted for clarity.)

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

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, $P_I = P_{II} = 0.5$ 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 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 a search 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 cluster 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 now 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:

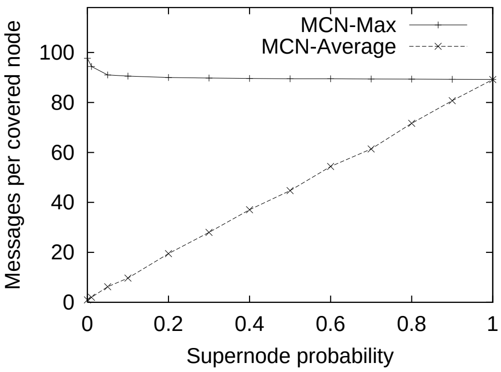


Figure 5: Tuning supernode networks.

- *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 to supernode networks, but we must first decide how many supernodes there will be in the network. We ran an experiment where we varied P_{sn}, the probability that a node was chosen as a supernode, from 0 to 1. 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. 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. Because supernodes are more heavily loaded than normal nodes, as the number of supernodes increases more nodes in the network are heavily loaded and the MCN-Average goes up. At the same time, as the number of supernodes increases, MCN-Max initially decreases, and then levels off. When there are more supernodes, the index update load from normal nodes is spread around to more nodes, and the MCN-Max decreases as the update load on supernodes decreases. However, every search message is processed by every supernode, and the MCN-Max levels off at about 90 messages per unit time, which represents search load that cannot be eliminated by changing 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

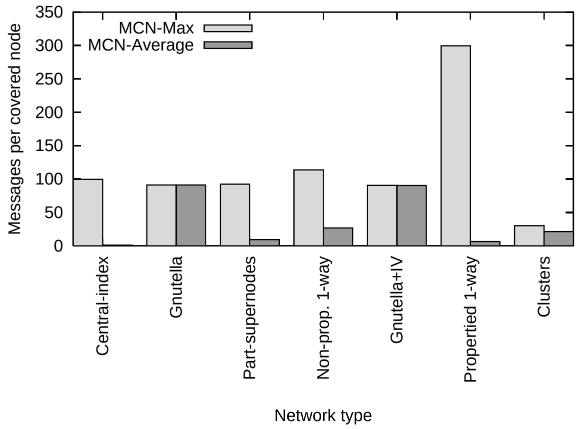


Figure 6: Ad hoc connect() networks compared to existing networks.

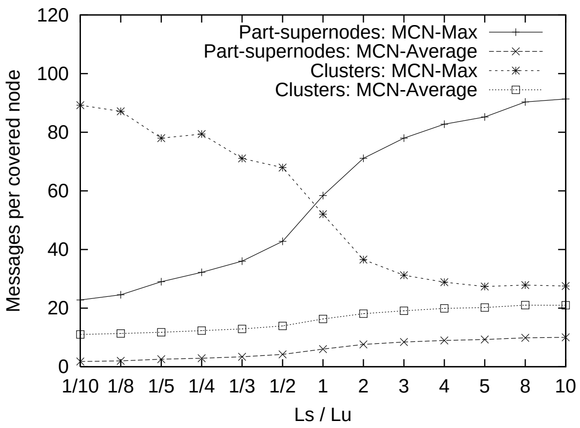


Figure 7: Cluster networks versus supernode networks for various load patterns.

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. Figure 6, which represents $L_S >> L_U$, shows these supernode networks compared to networks built using our ad hoc techniques. Note that MCN-Average for central-indexing is 0.993, not zero.

First, note that the ad hoc networks tend to have equal or higher MCN-Max than the supernode networks. Even if the MCN-Average of these networks is low, the MCN-Average of the central-index network is still lower, meaning that the ad hoc networks do not represent an improvement over supernodes.

The exception is the clusters network. This network has a significantly lower MCN-Max than any of the supernode networks. For example, the Gnutella network has an MCN-Max of 91.1 messages per covered node. This is roughly the lowest MCN-Max that is achievable in a supernode network, since in any supernode network there would be at least one node handling all of the searches, and this node would have an MCN-Max of at least 90. In comparison, the clusters network has an MCN-Max of only 30.1 messages per covered node. This indicates that clusters is more appropriate for networks where there are few or no nodes capable of handling the high load of supernodes. The increase in MCN-Average over part-supernodes indicates that in the cluster topology the load is being spread around the network more effectively, so that every node is contributing search resources and no one node is significantly overloaded. 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.

Figure 7 illustrates how cluster networks achieve better load balancing (significantly lower MCN-Max) 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 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 graphs for each connect() type. Our simulations involved over three thousand combinations of network type, break method, B_I and B_T values. Due to space limitations, the details of this optimization are omitted. 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 tried both propertied and non-propertied connect() operations. Some representative results from the 28 network types we tried are shown in Figure 8 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.

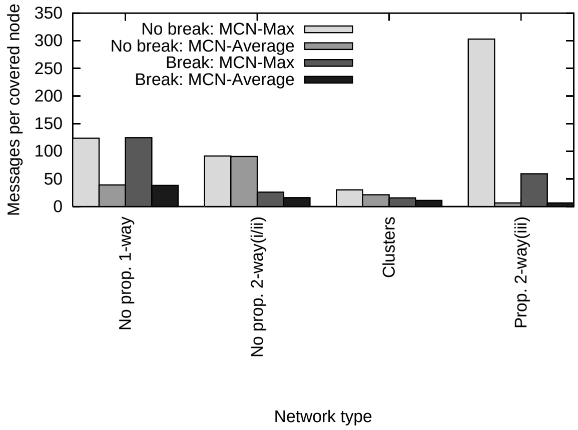


Figure 8: The break() operation versus no break().

One type of network that did benefit from break() is the non-propertied 2-way(I/II) network, with 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 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).

In contrast, 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 8. 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 cluster 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 $L_S << L_U$) 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.

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.

4.3.1 Comparison to supernode networks

Figure 9 shows the comparison between connect()/break() ad hoc networks and supernode networks for the $L_S >> L_U$ scenario. As this figure shows, many of 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 cluster network has about

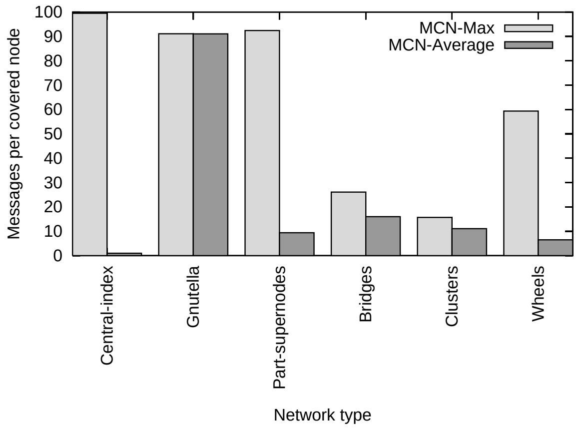


Figure 9: Ad hoc connect() and break() networks compared to supernode networks.

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 cluster 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 cluster 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.

5 Related work

Many researchers have focused on the problem of peer-to-peer search networks. Several studies [15, 14] have focused on characterizing networks such as Gnutella, and many researchers agree that a flooding network like Gnutella cannot easily scale. Pandurangan et al have looked at techniques for constructing Gnutella-like networks with desirable properties [12], 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 [6], local indexing networks [17], and of course supernode networks [2, 18].

Some researchers have abandonded flooding networks in favor of highly structured distributed hash tables (DHTs), such as Chord [16] and CAN [13], 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 [13]). The search networks we study here combine the operations of discovery ("what information is available?") and location ("where is a particular file located?"). Moreover, DHTs are not yet widely deployed, and the popularity of Gnutella and supernode networks suggest that flooding networks are still worthy of study despite the advent of DHTs.

Still others have proposed to retain the unstructured nature of search networks but to replace the flooding-based searching with random walk searches [10]. In these systems, content is proactively replicated throughout the network to ensure that the random walks can find content [4], and replication may place an undue burden on sites that store data which they have no interest in. Moreover, random walks may travel many nodes before finding content, which increases the latency of searches.

The concept of self-supervising networks as we use it is similar to Dijkstra's concept of a self-stabilizing system [7] in that the system converges to a desirable state through the actions of the nodes in the system. Self-supervision has been examined for peer-to-peer systems specifically. For example, Anthill [3] is a framework for developing self-organizing and self-tuning networks, although the techniques used (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 [9], privacy-preserving publishing [8], and content discovery in distributed digital archives [5]. Our techniques can be used to provide efficient and adaptive information discovery services for these applications.

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 forms and changes as necessary 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.

References

[1] Gnutella. http://gnutella.wego.com, 2003.

[2] Kazaa. http://www.kazaa.com, 2003.

[3] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A framework for the developement of agent-based peer-to-peer systems. In *Proc. of 22nd ICDCS*, 2002.

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

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

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

[7] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *CACM*, 17(11):643–644, Nov. 1974.

[8] R. Dingledine, M.J. Freedman, and D. Molnar. The FreeHaven Project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[9] J. Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Nov. 2000.

[10] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of ACM International Conference on Supercomputing (ICS'02)*, June 2002.

[11] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make gnutella scalable? In *Proc. of the 1st Int'l Workshop on Peer to Peer Systems (IPTPS)*, March 2002.

[12] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter P2P networks. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2001.

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

[14] M. Ripeanu and I. Foster. 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)*, March 2002.

[15] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Conferencing and Networking*, Jan. 2002.

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

[17] B. Yang and H. Garcia-Molina. Efficient search in peer-to-peer networks. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, July 2002.

[18] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proc. International Conference on Data Engineering*, 2003.