Search and Replication in Unstructured Peer-to-Peer Networks

Qin Lv Pei Cao Edith Cohen Kai Li Scott Shenker

Abstract

Decentralized and unstructured peer-to-peer networks such as Gnutella are attractive for certain applications because they require no centralized directories and no precise control over network topologies and data placement. However, the flooding-based query algorithm used in Gnutella does not scale; each individual query generates a large amount of traffic and, as it grows, the system quickly becomes overwhelmed with the query-induced load. This paper explores, through simulation, various alternatives to gnutella's query algorithm, data replication method, and network topology. We propose a query algorithm based on multiple random walks that resolves queries almost as quickly as gnutella's flooding method while reducing the network traffic by two orders of magnitude in many cases. We also present a distributed replication strategy that yields close-to-optimal performance. Finally, we find that among the various network topologies we consider, uniform random graphs yield the best performance.

1 Introduction

The computer science community has become accustomed to the Internet's continuing rapid growth, but even to such jaded observers the explosive increase in Peer-to-Peer (P2P) network usage has been astounding. Within a few months of Napster's [12] introduction in 1999 the system had spread widely, and recent measurement data suggests that P2P applications are having a very significant and rapidly growing impact on Internet traffic [11, 15]. Therefore, it is important to study the performance and scalability of these P2P networks.

Currently, there are several different architectures for P2P networks:

Centralized: Napster and other similar systems have a constantly-updated directory hosted at central locations (*i.e.*, the Napster web site). Nodes in the P2P network issue queries to the central directory server to find which other nodes hold the desired files. While Napster was extremely successful before its recent legal troubles, it is clear that such centralized approaches scale poorly and have single points of failure.

Decentralized but Structured: These systems have no central directory server, and so are decentralized, but have a significant amount of structure. By "structure" we mean that the P2P network topology (that is, the set of connections between P2P members) is tightly controlled and that files are placed not at random nodes but at specified locations that will make subsequent queries easier to satisfy. In some systems, those with "loose structure," this placement of files is based on hints; the Freenet P2P network [8] is an example of such a system. In systems with "tight structure", the structure of the P2P network and the placement of files is extremely precise and so subsequent queries can be satisfied very efficiently. There is a growing literature on what might be called *lookup systems* which support a hash-table-like interface; see [16, 20, 18, 23]. Such tightly structured P2P designs are quite prevalent in the research literature, but almost completely invisible on the current network. Moreover, it isn't clear how well such designs work with an extremely transient population of nodes, which seems to be a characteristic of the Napster community.

Decentralized and Unstructured: These are systems in which there is neither a centralized directory nor any precise control over the network topology or file placement. Gnutella [9] is an example of such designs. The network is formed by nodes joining the network following simple loose rules (for example, those described in [6]). The resultant topology has certain properties, but the placement of files is not based on any knowledge of the topology (as it is in structured designs). To find a file, a node queries its neighbors. The most typical query method is flooding, where the query is propogated to all neighbors within a certain radius [6]. These unstructured designs are extremely resilient to nodes entering and leaving the system. However, the current search mechanisms are extremely unscalable, generating large loads on the network participants.

In this paper, we focus on gnutella-like decentralized, unstructured P2P systems. We do so because (1) these systems are actively used by a large community of Internet users [22, 7], and (2) these systems have not yet been subject to much serious research, except for empirical studies.

The goal of this paper is to study more-scalable alternatives to existing Gnutella algorithms, focusing on the search and replication aspects. We first quantify the poor scaling properties of the flooding search algorithms. We then propose, as an alternative, a k-walker random walk algorithm that greatly reduces the load generated by each query. We also show that active replication (where the files may be stored at arbitrary nodes) produces lower overall query load than non-active node-based replication (*i.e.*, a file is only replicated at nodes requesting the file). Path replication, where the file is replicated along the path from the requester to the destination, yields a close-to-optimal replication distribution. Finally, we show that for unstructured networks, power-law random graphs are less desirable than uniform random graphs and so P2P system should adopt graph-building algorithms that reduce the likelihood of very-high degree nodes.

This paper has 8 sections. In Section 2 we describe our model of unstructured P2P systems and our evaluation methodology. In Section 3 we discuss the limitations of the flooding approach currently used by gnutella, and then, in Section 4, we propose and evaluate various alternative approaches. We discuss replication strategies in Section 5 and evaluate these approaches in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 Modeling and Evaluation Methodology

It is impossible to model all the dynamics in a P2P system on the Internet. Hence, we use simple models to derive understanding of the behavior of different algorithms, and verify the behavior of the algorithms on slightly more realistic simulations. We are not looking for quantitative results, but rather qualitative ones, so we omit many details in our modeling and simulation.

2.1 Abstractions

We look at three aspects of a P2P system: network topology, query distribution and replication distribution. By network topology, we mean the graph formed by nodes in the network at an instant. For simplicity, we assume that the graph does not change during the simulation of our algorithms. By query distribution, we mean the distribution of frequency of lookups to files. Again, we assume that this distribution is fixed. By replication distribution, we mean the distribution of the percentage of nodes that have a particular file. During our study of search algorithms, we assume static replication distributions.

Our assumption of fixed network topology and fixed query distributions are obviously gross simplifications. However, for the purpose of our study, if one assumes that the time to complete a search is short compared to the time of change in network topology and change in query distribution, results obtained from these settings are still indicative of performance in real systems.

We use four network topologies in our study:

- Power-Law Random Graph (PLRG): this is a 9230-node random graph. The node degrees follow a power-law distribution; if one ranks all nodes from the most connected to the least connected, then the i'th most connected node has ω/i^{α} neighbors, where ω is a constant. Many real-life P2P networks have topologies that are power-law random graphs[13]. This particular graph has $\alpha = 0.8$. The graph is generated by a modified version of the GT-ITM topology generator [21].

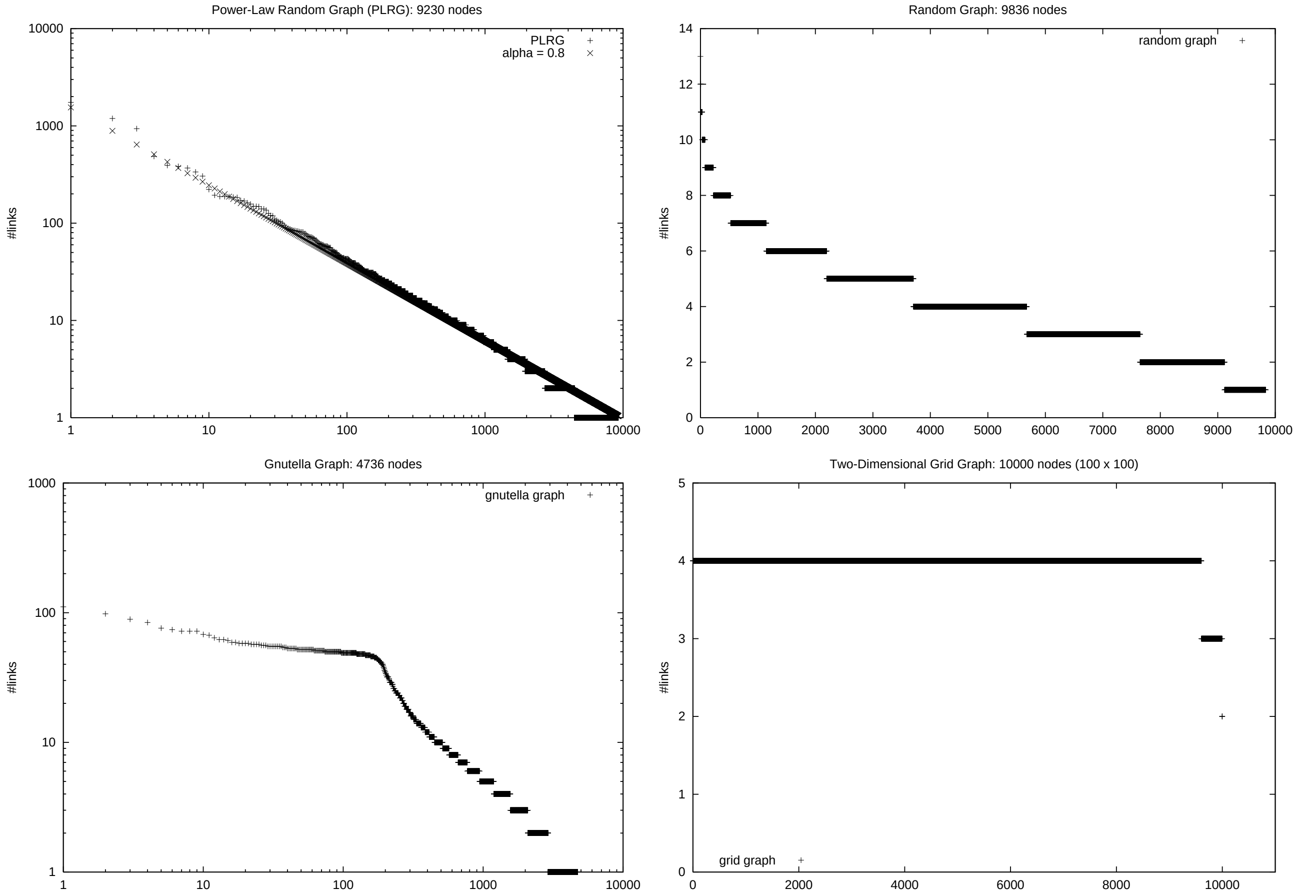


Figure 1: Distribution of node degrees in the four network topology graphs. Note that we use log scale for PLRG and Gnutella, and linear scale for Random and Grid.

- Normal Random Graph (Random): a 9836-node random graph generated by a modified version of GT-ITM topology generator[21]. The average degree of nodes in the graph is 4. The distribution is shown in Figure 1.
- Gnutella graph (Gnutella): a snapshot of the gnutella P2P network in August 2001 [4]. The graph has 4736 nodes. Its node degrees roughly follow a two-segment power-law distribution, as shown in Figure 1.
- Two-Dimensional Grid (Grid): a 10000-node two-dimension grid (100x100). We choose this simple graph for comparison purposes.

Node degree distributions of the four graphs are shown in Figure 1. Key statistics of the graphs are also summarized in Table 2.1.

	#nodes	total #edges	avg. node degree	std. dev.	max degree	median degree
PLRG	9230	20599	4.46	27.9	1746	1
Random	9836	20099	4.09	1.95	13	4
Gnutella	4736	13022	5.50	10.7	136	2
Grid	10000	19800	3.96	0.20	4	4

Table 1: Key statistics of the network topologies that we used.

For query distribution, we assume that there are N objects of interest. (In this paper we use the terms "file" and "object" interchangeably.) We investigate the following distributions:

- Uniform: all objects are equally popular. In other words, the probability that an object will be searched by a node in the network is $1/N$, where N is the number of objects of interest.
- Zipf-like: object popularity follows a Zipf-like distribution. If one ranks all objects from the most popular to the least popular, then the probability that the i'th ranked object is searched by a node is proportional to $1/i^{\alpha}$. Studies have shown that Napster, Gnutella and Web queries tend to follow Zipf-like distributions [3, 19].

In other words, our query distributions are:

$$\begin{aligned} \sum_{i=1}^{N} q_i &= 1 && (1)\\ Uniform: q_i &= \frac{1}{N} && (2)\\ Zipf-like: q_i &\propto i^{-\alpha} && (3) \end{aligned}$$

For replication distribution, we simulate it in two ways. In the first part of our study, where we focus on search methods, we assume **static** replication, where object i is randomly placed at r_i nodes, with r_i following certain distribution. Since we study unstructured networks, when an object is replicated at r_i nodes, the r_i nodes are randomly choosen in the graph. In the second part of the study, where we study different replication strategies and their impact on search efficiency, we simulate the dynamic replication of the objects.

We use the following static replication distributions:

- Uniform: all objects are replicated at roughly the same number of nodes. In the simulation, we assume that each object is replicated at 1% of the nodes.
- Proportional: the replication ratio of an object i is proportional to the query probability of the object. If one assumes only nodes requesting an object store the object, then the replication distribution is usually proportional to query distribution.
- Square-root: replication ratio of an object i is proportional to the square root of its query probability q_i. The reason for the square-root distribution is discussed in Section 5.

In other words, our replication distributions are:

$$\begin{aligned} \sum_{i=1}^{N} r_i &= R \times N && (4)\\ Uniform: r_i &= R && (5)\\ Proportional: r_i &\propto q_i && (6)\\ Square-root: r_i &\propto q_i^{\frac{1}{2}} && (7) \end{aligned}$$

where R is the average replication ratio.

Clearly, there are three combinations of query distribution and replication distribution: uniform/uniform, zipf/proportional, and zipf/square-root.

2.2 Metrics

Performance issues in real P2P systems are extremely complicated. In addition to issues such as load on the network, load on network participants, and delays in getting positive answers, there are a host of other criteria such as success rate of the search, the bandwidth of the selected provider nodes, and fairness to both the requester and the provider. It is impossible for us to use all of these criteria in evaluating search and replication algorithms.

Instead, we focus on efficiency aspects of the algorithms solely, and use the following simple metrics in our abstract P2P networks. These metrics, though simple, reflect the fundamental properties of the aglorithms.

- User aspects:
  - Pr(success): the probability of success of finding the queried object before the search terminates. Different algorithms have different criteria for terminating the search, and lead to different probability of success under various replication distributions.
  - #hops: delay in finding an object as measured in number of hops. We do not model the actual network latency here, but rather just measure the abstract number of hops that a successful search message travels before it replies to the originator.
- Load aspects:
  - avg. #msgs per node: overhead of an algorithm as measured in average number of search messages each node in the network has to process. The motivation for this metrics is that in P2P systems, the most notable overhead tend to be the processing load that the network imposes on each participant. The load, usually interrupt processing or message processing, is directly proportional to the number of messages that the node has to process.
  - #nodes visited: the number of network participants that a query's search messages travel through. This is an indirect measure of the impact that a query generates on the whole network,
  - peak #msgs: to identify hot spots in the network, we calculate the number of messages that the busiest node has to process for a set of queries.
- Aggregate performance: for each of the above measures, which are per-query measures, we calculate an aggregate performance number, which is the performance of each query convoluted with the query probability. That is, for each object i, under each network settings, if the performance measure is $a(i)$, then the aggregate is sum($q_i * a(i)$).

2.3 Simulation Methodology

In our evalution of each search method, we run a set of simulations for each combination of query distribution and replicatin distribution, and report results averaged over the set of simulations. In all the simulations, the number of objects, N, is 100, the average replication ratio, R, is 1.0%, and the parameter α in the Zipf-like distribution is 1.20.

For each set of simulations, we first select the topology file. We then generate *numPlace* sets of replica placements; replicas are are placed *randomly* in the network, and the distribution of replicas follows the specified replication distriubtion. For each replica placement, we generate *numQuery* different queries following the specified query distribution. Then for each of the queries, we simulate the searching process using the designated search method. We can run the simulation for each query independent of other queries because the object replication is fixed, and hence running all the queries concurrently is the same as running each one separately and then summing the results.

Statistics are collected from the *numPlace* $\times$ *numQuery* queries. In each set of simulations, *numPlace = 10* and *numQuery = 100*. This results in 1000 different queries. We then calculate the results in the following way: *Pr(success)* is the number of successful queries divided by the total number of queries generated; *avgHops* is the average number of hops taken for each successful query; *avgMsgs* is the average number of messages generated for

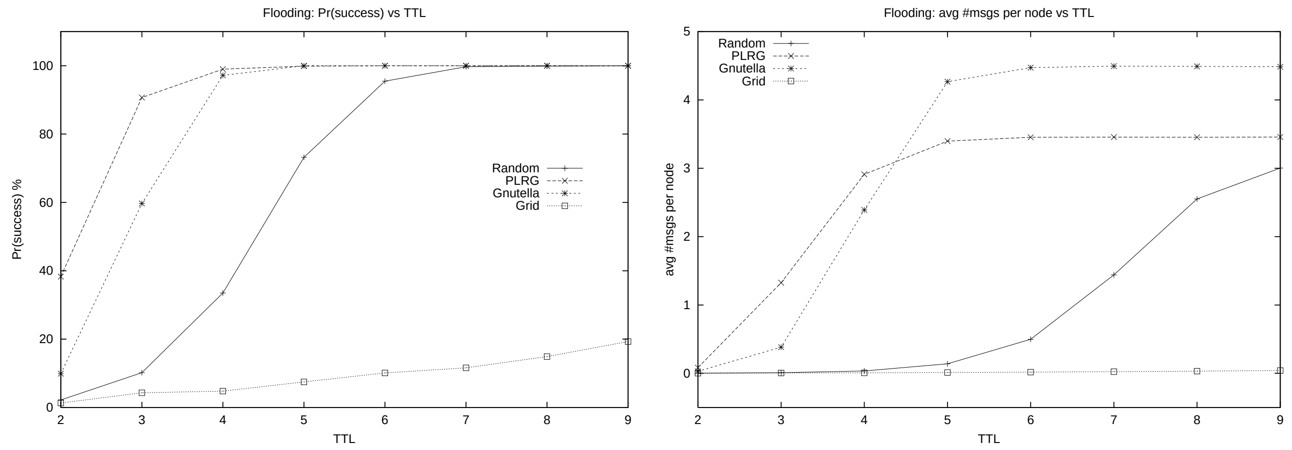


Figure 2: How the probability of success and the per-node message overhead vary by the TTL setting in four network topologies.

each query divided by the total number of nodes in that network; *avgNodes* is the average number of nodes visited (i.e. received at least one message) during each query; *dupMsgs* is the average duplication percentage calcualted as *(msgs - nodes)/msgs*; and *peakMsg* is the maximal sum of messages received by any single node during the set of simuliations, divided by the total number of queries in that set of simulation. These averages are presented in tables and figures in this paper. Due to space limitation we do not present standard deviations associated with the averages, except to point out that the standard deviation data do not change the conclusions in the paper. We plan to present all the data in a web site later.

As a final note about our abstractions and metrics, we stress that they omit a lot of issues, including the true dynamics of node coming and going in the network, the message delays in the network, the actual load on a network node for processing and propogating messages, etc. However, these models help us understand the fundamental properties of various search and replication algorithms.

3 Limitations of Flooding

One of the major load isssues in P2P networks is the load on individual network participants. Typically, the participants are PCs at home or office, and are used for normal work and entertainment. If a PC has to handle many network interrupts when it joins the P2P network, the user will be forced to take the PC off the P2P network to get “real” work done. This in turn limits the size and the usefulness of the P2P network.

Unfortunately, the simple flooding-style search used in Gnutella exacerbates this problem. There are two issues with flooding. First, it is difficult to choose an appropriate Time-To-Live (TTL) to terminate the flood. Typically, each query has a TTL assigned to it at the beginning. Each time it is propagated from one node to the node’s neighbors, the TTL is reduced by 1. The process continues until TTL reaches 0. For a node that wants to find an object but does not know how widely replicated the object is, picking the right TTL is tricky. If the TTL is too high, the node unnecessarily burdens the network. If the TTL is too low, the node might not find the object even though a copy exists somewhere.

To illustrate the problem, Figure 2 shows the probability of success and average per-node message overhead of flooding as TTL increases. The search is for an object that is only replicated at 0.125% of the nodes, which means that on average, 800 nodes need to be visited to find the object. We can see from the figures that different TTLs are needed to reach this coverage in different network topologies. Unfortunately, since in practice the replication ratio of an object is unknown, users have to set TTLs high to ensure success of the query.

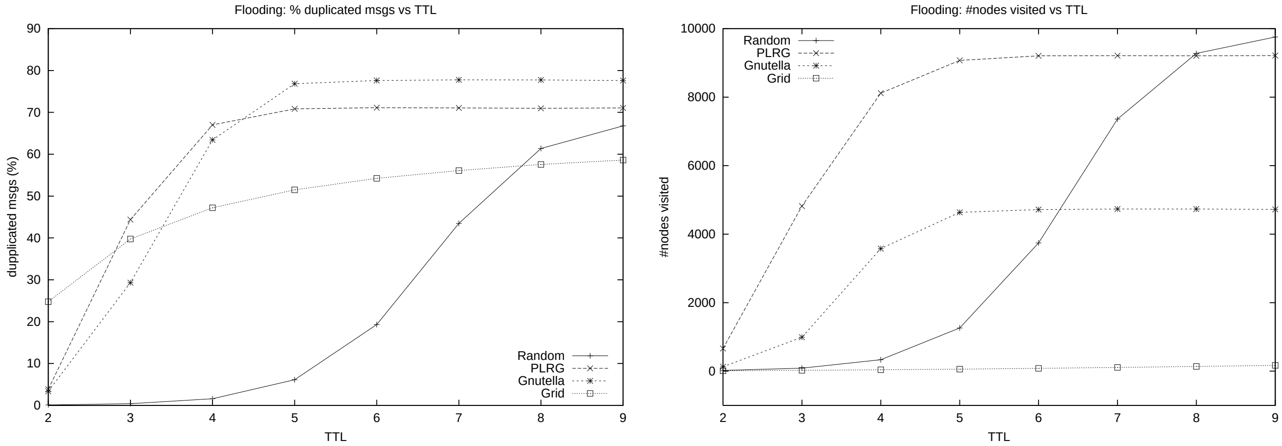


Figure 3: As the TTL increases, the percentage of query messages that are duplicates increases because a node has more neighbors forwarding the messages to it.

Second, there are many duplicate messages introduced by flooding, particularly in high connectivity graphs. By duplicate messages we mean the multiple copies of a query that are sent to a node by its multiple neighbors. Flood-style search does have duplication detection built in, requiring the node to detect and discard duplicate queries. However, duplicate queries are pure overhead in flooding. They incur extra network interrupt processing at the nodes receiving them, and do not contribute to increased chance of finding the object. The problem worsens as the TTL increases.

Figure 3 shows the percentage of duplicate messages and the number of unqiue nodes visited as TTL increases. As we can see from the graphs, when TTL increases, the number of unique nodes visited increases, but at the same time, the percentage of duplicate messages also increases. In other words, in flooding, it is not possible to increase the number of nodes covered in a search without increasing the duplication in the search.

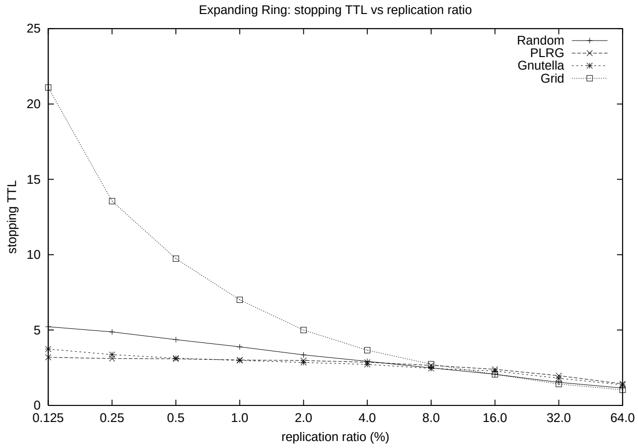
These limitations mean that flooding incurs considerable message processing overhead for each query, increasing the load on each node as the network expands and the query rate increases, to the point that a node can be so loaded that it has to leave the network. Other researchers have also noted the limitations of flooding [17].

Our simulations also show that Power-Law random graphs and Gnutella style graphs are particularly bad with flooding. The presence of the highly connected nodes mean that its duplication ratios are much higher than those in the random graph, because many nodes' neighbors overlap. In fact, for flooding, the random graph would be the best topology, because in a true random graph, the duplication ratio (liklihood that the next node already recieved the query) is the same as the fraction of nodes visited so far, as long as that fraction is small.

The random graph is also better for load distribution among its nodes. In the random graph, the maximum load on any one node is logarithmic to the total number of nodes that the search visits. In contrast, the high degree nodes in PLRG and Gnutella graphs have much higher load than other nodes. Due to space constraints we omit the data on peak # of messages here.

4 Finding Better Search Methods

Since flooding has inherent limitations, we try to find more scalable search methods for unstructured networks. Our first try is aimed at addressing the TTL selection problem.



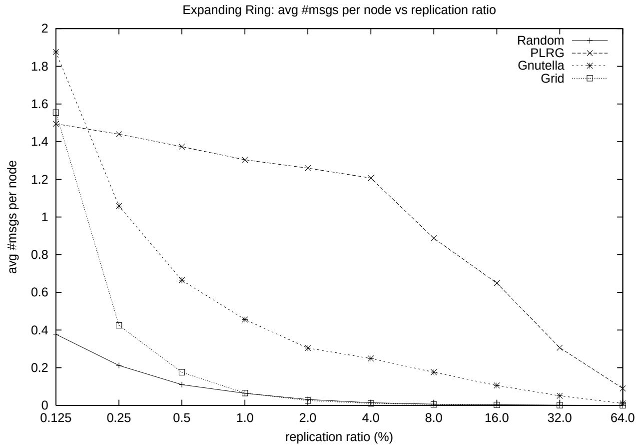


Figure 4: The stopping TTL and the per-node message overhead for expanding ring when searching objects of various replication ratio.

4.1 Expanding Ring

One might be tempted to solve the TTL selection problem by asking nodes to check with the original requester before forwarding the query to neighbors. This way, the flood can be called off when the object is found. However, the approach can lead to message implosion at the requester node. Hence, we do not adopt this approach.

Instead, we use successive floods with increasing TTLs. A node starts a flood with small TTL, and waits to see if the search is successful. If it is, then the node stops. Otherwise, the node increases the TTL and starts another flood. The process repeats until the object is found. We expect this method to perform particularly well when hot objects are replicated more widely than cold objects, which is likely the case in practice. We call this method "expanding ring."

To understand how well expanding ring works, we measure the average stopping TTL for searches to objects with increasing replication ratios. In our simulations the expanding ring starts with $TTL = 1$, and expands the TTL linearly by 2 each time. Figure 4 shows the results for various topologies. As we can see, expanding ring successfully reins in the TTL as the object's replication ratio increases. While searches for objects with low replication ratio need TTLs larger than 5, the searches stop at TTL of 1 or 2 when the object replication is over 10%.

However, this adaptivity does not necessarily translate to lower message overhead, because the successive retries could lead to more messages. To understand the message overhead of expanding ring, we also record the average number of messages a node has to process. The results are shown in Figure 4. Comparing the message overhead results between flooding and expanding ring, we can see that, for objects that are replicated at .125% of the nodes, even if flooding uses the best TTL for each network topology, expanding ring still halfs the per-node message overhead.

To understand the overall impact of expanding rings in a P2P network, we simulate the completion of 1000 queries in P2P networks with different combinations of query distribution and replication distribution. The results are shown in the "expanding ring" column in Tables 2 through 5. (The tables are a comparison of various performance metrics of search methods in the four network topologies.)

The results show that, despite the successive retries, expanding ring still reduces message overhead significantly compared with regular flooding with a fixed TTL. The savings are obtained across all query and replication distributions, even for uniform replication distributions. The improvements are also more pronounced for Random and Gnutella graphs than for PLRG graph, partly because the very high degree nodes in PLRG graph reduce the oppurtunity for incremental retries in expanding ring.

Expanding ring achieves the savings at the expense of slight increase in the delays to find the object. Since we do not simulate actual network latency but use number of hops to estimate the latency, the tables also include a row

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	walk	state	flood	ring	walk	state
Uniform / Uniform	#hops	3.40	5.77	10.30	7.00	3.40	5.77	10.30	7.00
	#msgs per node	2.509	0.062	0.031	0.024	2.509	0.061	0.031	0.024
	#nodes visited	9220	536	149	163	9220	536	149	163
	peak msgs	6.37	0.26	0.22	0.19	6.37	0.26	0.22	0.19
Zipf-like / Proportional	#hops	1.60	2.08	1.72	1.64	2.51	4.03	9.12	6.66
	#msgs per node	1.265	0.004	0.010	0.010	1.863	0.053	0.027	0.022
	#nodes visited	6515	36	33	47	7847	396	132	150
	peak msgs	4.01	0.02	0.11	0.10	5.23	0.20	0.17	0.14
Zipf-like / Square root	#hops	2.23	3.19	2.82	2.51	2.70	4.24	5.74	4.43
	#msgs per node	2.154	0.010	0.014	0.013	2.308	0.031	0.021	0.018
	#nodes visited	8780	92	50	69	8983	269	89	109
	peak msgs	5.88	0.04	0.16	0.16	6.14	0.12	0.17	0.16

Table 2: Simulation results of search methods for Random Graph. The first set of columns are results for queries to the top 50% of hottest objects; the second set of columns are results for all queries. "Flood" is flooding with TTL=8. "Ring" is expanding ring. "Walk" is 32-walker random walk with checking. "State" is 32-walker random walk with checking and state keeping.

on average number of hops as an indication of user-experienced delay. As we can see, for Random, PLRG and Gnutella, the average # of hops only increases from 2 to 4 in flooding to 3 to 6 in expanding ring, which we believe is tolerable for users.

Though expanding ring solves the TTL selection problem, it does not address the message duplication issue inherent in flooding. Inspection of simulation results shows that the duplication contributes significantly to the message overhead, particularly for PLRG and Gnutella graphs. To reduce message duplication, we try a different approach, random walk.

4.2 Random Walks

Random walk is a well-known technique. To search for an object using random walk, a node chooses a neighbor randomly and sends the query to it. The neighbor in turns chooses one of its neighbors randomly and forwards the query. The process continues until the object is found. For each query, only one copy of it is present in the network. We call the copy a "walker."

Our initial attempt is to use the standard random walk as a search method. As expected, random walk cuts down the message overhead significantly, by an order of magnitude compared to expanding ring across the network topologies. However, this efficiency comes at an order of magnitude increase in user-perceived delay of successful searches.

Investigations show that the key to reducing this delay is to reach the desired number of nodes as quickly as possible. Hence, we decide to increase the number of "walkers" in the random walk. That is, instead of just sending out one query message, a requesting node sends N query messages, and each query message takes its own random walk. The expectation is that N walkers after T steps should reach the same number of nodes as 1 walker after $N * T$ steps, and indeed simulations confirm that. Therefore, by using N walkers, we can expect to cut the delay down by a factor of N.

Since multiple-walker random walks require a mechanism to terminate the walks, we experimented with two methods, TTL and "checking.". TTL means that, similar to flooding, each random walk terminates after a certain

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	walk	state	flood	ring	walk	state
Uniform / Uniform	#hops	2.37	3.50	8.95	8.47	2.37	3.50	8.95	8.47
	#msgs per node	3.331	1.325	0.030	0.029	3.331	1.325	0.030	0.029
	#nodes visited	8935	4874	147	158	8935	4874	147	158
	peak msgs	510.4	132.7	12.3	11.7	510.4	132.7	12.3	11.7
Zipf-like / Proportional	#hops	1.74	2.36	1.81	1.82	2.07	2.93	9.85	8.98
	#msgs per node	2.397	0.593	0.011	0.011	2.850	0.961	0.031	0.029
	#nodes visited	6969	2432	43	49	7923	3631	136	145
	peak msgs	412.7	58.3	4.9	5.1	464.3	98.9	12.7	11.7
Zipf-like / Square root	#hops	2.07	2.94	2.65	2.49	2.21	3.17	5.37	4.79
	#msgs per node	3.079	0.967	0.014	0.014	3.199	1.115	0.021	0.020
	#nodes visited	8434	3750	62	69	8674	4200	97	103
	peak msgs	496.0	93.7	6.3	6.3	499.6	111.7	8.9	8.4

Table 3: Simulation results of search methods for Power-Law Random Graph (PLRG). The legends are the same as in Table 2.

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	walk	state	flood	ring	walk	state
Uniform / Uniform	#hops	2.39	3.40	7.30	6.11	2.39	3.40	7.30	6.11
	#msgs per node	4.162	0.369	0.051	0.045	4.162	0.369	0.051	0.045
	#nodes visited	4556	933	141	151	4556	933	141	151
	peak msgs	64.9	6.4	1.3	1.2	64.9	6.4	1.3	1.2
Zipf-like / Proportional	#hops	1.60	2.18	1.66	1.66	2.03	3.05	9.39	7.94
	#msgs per node	2.961	0.109	0.021	0.021	3.548	0.423	0.058	0.051
	#nodes visited	3725	357	49	60	4137	810	143	153
	peak msgs	43.8	2.0	0.7	0.8	54.5	7.0	1.6	1.5
Zipf-like / Square root	#hops	1.88	2.70	2.31	2.15	2.10	3.02	4.61	4.12
	#msgs per node	3.874	0.208	0.027	0.026	4.007	0.302	0.038	0.035
	#nodes visited	4404	621	67	80	4479	789	101	114
	peak msgs	62.5	3.8	0.8	0.9	63.8	5.3	1.1	1.1

Table 4: Simulation results of search methods for Gnutella Graph. The legends are the same as in Table 2.

distribution model		50 % (queries for hot objects)				100 % (all queries)			
query/replication	metrics	flood	ring	walk	state	flood	ring	walk	state
Uniform / Uniform	#hops	6.52	19.15	27.95	15.20	6.52	19.15	27.95	15.20
	#msgs per node	0.472	0.070	0.068	0.041	0.472	0.070	0.068	0.041
	#nodes visited	1692	128	107	128	1692	128	107	128
	peak msgs	0.72	0.18	0.30	0.16	0.72	0.18	0.30	0.16
Zipf-like / Proportional	#hops	1.70	2.32	1.95	1.77	4.71	19.04	33.78	15.14
	#msgs per node	0.321	0.003	0.011	0.010	0.392	0.120	0.082	0.040
	#nodes visited	1398	14	22	28	1533	118	111	121
	peak msgs	0.57	0.02	0.07	0.06	0.64	0.25	0.26	0.16
Zipf-like / Square root	#hops	2.77	4.64	4.60	3.32	4.31	10.66	15.53	8.22
	#msgs per node	0.437	0.008	0.018	0.015	0.450	0.034	0.041	0.025
	#nodes visited	1647	31	34	42	1656	70	67	77
	peak msgs	0.68	0.04	0.10	0.08	0.68	0.10	0.19	0.12

Table 5: Simulation results of search methods for Grid Graph. The legends are the same as in Table 2.

number of hops. "Checking" means that a walker periodically checks with the original requester before walking to the next node (of course, the checking is actually done by the node forwarding the walker). The checking method still uses a TTL, but the TTL is very large and is mainly used to prevent loops.

Our simulations show that checking is the right approach for terminating searches in random walks. The TTL approach runs into the same TTL selection issue in flooding. Meanwhile, since there are a fixed number of walkers (typically 16 to 64), having the walkers check back with the requester will not lead to message implosion at the requester node. Of course, checking does have overhead; each check requires a message exchange between a node and the requester node. Further experiments show that checking once every fourth step along the way strikes a good balance between the overhead of the checking messages and the benefits of checking.

We experimented with different number of walkers. With more walkers, we can find objects faster, but also generate more loads. And when the number of walkers is big enough, increasing it further yield little reduction in the number of hops, but significantly increases the message traffic. Usually, 16 to 64 walkers give good results. We choose 32 walkers in our simulations. (Due to space limitations we omit the results here.)

Tables 2 through 5 compare the discussed search methods under all combinations of query and replication distributions for the four network topologies. There are two sets of columns in each table; the first set are results of the queries to 50th percentile of hottest objects, and the second set are results of all queries.

The results show that the 32-walker random walk reduces message overhead by *two orders of magnitude* for all queries across all network topologies, at the expense of slight increase in the number of hops (increasing from 2-6 to 7-15). The 32-walker random walk generally outperforms expanding ring as well, particularly in PLRG and Gnutella graphs.

We also studied an improvement to the above approach by asking each node to keep states. When a search is started, all N walkers are tagged with a unique ID. For each ID, a node remembers the neighbors to which it has forwarded queries of that ID, and when a new query with the same ID arrives, the node forwards it to a different neighbor (randomly chosen). This state keeping accelerates the walks because walkers are less likely to cover the same route and hence they visit more nodes. Simulation results, also shown in the tables, confirm the improvement. Compared with random walks without state keeping, random walk with state keeping shows the biggest improvement in Random and Grid graphs, reducing message overhead by up to 30%, and reducing number of hops by up to 30%. However, the improvements for PLRG and Gnutella graphs are small. Hence, depending on the implementation overhead of state keeping, each P2P network should decide separately whether state keeping is worthwhile.

4.3 Principles of Scalable Searches in Unstructured Networks

Our results show that the k-walker random walk is a much more scalable search method than flooding. However, perhaps more important than this conclusion is the understanding we have gained from this exercise. We summarize it here.

The key to scalable searches in unstructured network is to cover the right number of nodes as quickly as possible and with as little overhead as possible. In unstructured network, the only way to find objects is to visit enough nodes so that, statistically speaking, one of the nodes has the object. However, in reaching the required node coverage, one must pay attention to the following:

- *Adaptive termination is very important.* TTL-based mechanism does not work. Any adaptive/dynamic termination mechanism must avoid the implosion problem at the requester node. The checking method described above is a good example of adaptive termination.
- *Message duplication should be minimized.* Preferrably, each query should visit a node just once. More visits are wasteful in terms of the message overhead.
- *Granularity of the coverage should be small.* Each additional step in the search should not significantly increase the number of nodes visited. This perhaps is the fundamental difference between flooding and multiple-walker random walk. In flooding, an additional step could expoentially increase the number of nodes visited; in random walk, an additional step increases the number of nodes visited by a constant. Since each search only requires a certain number of nodes to be visited, the extra nodes covered by flooding merely increase the per-node load.

Under these constraints, a search algorithm should reduce the latency as much as possible.

We have not done an exhaustive study of all search algorithms, and we do not claim the k-walker random walk is optimal. However, we hope that the above principles will aid the understanding and search for the optimal methods.

5 Replication: Theory

Our study in the previous section examined how one should search for an object, assuming that it is replicated at some random locations in the network. Certain P2P systems such as Gnutella have rigid assumptions on how replications of objects happen in the system; that is, only nodes that request an object make copies of the object. Other P2P systems such as FreeNet allow for more proactive replications of objects, where an object may be replicated at a node even though the node has not requested the object.

For systems that allow proactive replications, we study the question: how many copies of each object should there be so that the search overhead for the object is minimized, assuming that the total amount of storage for objects in the network is fixed? Answers to this question have implications to non-proactive replication systems as well, because the information of an object's location could be proactively replicated to expedite the searches.

To formulate this question more precisely, we first use a very simple model to address the question theoretically. This model is more extensively analyzed in [5]. In the next section we use simulations to analyze the question.

We consider a simple model where there are n sites and m objects. Each object i is replicated at r_i random (distinct) sites, and set $R = \sum_i r_i$. We assume that the objects are requested with relative rates q_i, where we normalize this by setting $\sum_i q_i = 1$. For convenience, we assume that query and replication strategies are such that $1 \ll r_i \leq n$ and that searches go on until a copy is found. (The other cases are dealt with in [5], and the conclusions are consistent with, but a bit messier than, what we present here). Search consists of randomly probing sites until the desired object is found. Thus, the probability $Pr(k)$ that the object is found on the k'th probe is given by:

$$Pr_i(k) = \frac{r_i}{n}(1 - \frac{r_i}{n})^{k-1}$$

The *average search size* A_i is merely the fraction of sites which have replicas of the object:

$$A_i = \frac{n}{r_i}$$

We are interested in the average search size A, where $A = \sum_i q_i A_i = n \sum_i \frac{q_i}{r_i}$. The average search size essentially captures the message overhead of efficient searches.

If there were no limit on the r_i then clearly the optimal strategy would be to replicate everything everywhere, setting $r_i = n$, and then all searches become trivial. Instead, we assume that the average number of these replicas per site, $\rho = \frac{R}{n}$, is fixed and less than m. The question is how to allocate these R replicas among the sites.

The simplest replication strategy is to create the same number of replicas of each object: $r_i = \frac{R}{m}$. We call this the *uniform* replication strategy. In this case the average search size $A_{uniform}$ is given by:

$$A_{uniform} = \sum_i q_i \frac{m}{\rho} = \frac{m}{\rho}$$

which is independent of the query distribution.

It is very clear, though, that uniformly replicating all objects, even those that are not frequently queried, is inefficient. A more natural policy, one that results from having the querying sites cache the results of their query, is to replicate *proportional* to the querying rate: $r_i = Rq_i$. This should reduce the search sizes for the more popular objects.

However, a quick calculation reveals that the *average* remains the same:

$$A_{proportional} = n \sum_i \frac{q_i}{Rq_i} = \frac{m}{\rho} = A_{uniform}$$

Thus, the Proportional and Uniform replication strategies yield *exactly* the same average search size, and that average search size is independent of the query distribution.

Another important metric that captures the load balancing ability of a replication strategy is the *utilization rate*,

$$U_i = R\frac{q_i}{r_i}$$

that is, the rate of requests that a replica of object i serves (the random probing search process implies that all replicas of the same object have the same utilization rate). Note that the average utilization over all objects $U = \sum_i r_i U_i / R = 1$ is *fixed* for all replication strategies. The maximum utilization $\max_i U_i$, however, varies considerably.

The distributions of average search sizes and utilization rates for an object are quite different between the Uniform and Proportional strategies. For Uniform replication, all objects have the same average search size, but replicas have utilization rates *proportional* to their query rates. Proportional replication achieves perfect load balancing with all replicas having the same utilization rate, but average search sizes vary with more popular objects having smaller average search sizes than less popular ones. Objects whose query rates are greater than average (i.e., greater than $\frac{1}{m}$) do better with Proportional replication, and the other objects do better with Uniform replication. Interestingly, the weighted average of the search sizes over all objects balances out to be unchanged.

Square-Root Replication Given that Uniform and Proportional have the same average search size, a natural question is what is the optimal way to allocate the replicas so that the average search size is minimized? A simple calculation (see [5]) reveals that Square-Root replication is optimal; that is, A is minimized when $r_i = \lambda\sqrt{q_i}$ where $\lambda = \frac{R}{\sum_i \sqrt{q_i}}$. The average search size is

$$A_{optimal} = \frac{1}{\rho}(\sum_i \sqrt{q_i})^2$$

Table 6 lists properties of the three replication strategies. Square-Root replication is such that both average search size and utilization rate vary per object, but the variance in utilization is considerably smaller than with Uniform, and the variance in average search size is considerably smaller than with Proportional.

startegy	A	r_i	$A_i = n/r_i$	$U_i = Rq_i/r_i$
Uniform	$\rho^{-1}m$	R/m	$\rho^{-1}m$	$q_i m$
Proportional	$\rho^{-1}m$	$q_i R$	$(\rho q_i)^{-1}$	1
Square-Root	$\rho^{-1}(\sum_i \sqrt{q_i})^2$	$R\sqrt{q_i}/\sum_j \sqrt{q_j}$	$\rho^{-1}\sum_j \sqrt{q_j}/\sqrt{q_i}$	$\sqrt{q_i}\sum_j \sqrt{q_j}$

Table 6: Comparing the three replication strategies: Uniform, Proportional, and Square-Root.

5.1 Specific Query Distributions

We now consider query distributions $q_1 \geq q_2 \geq \cdots \geq q_m$ which are truncated Geometric and Pareto distributions. We compute the average search sizes $A_{optimal}$ for these distributions.

Truncated Geometric Distribution $G_m(\lambda)$ is defined by

$$q_i = \lambda^i/C \; (i = 1, \dots, m) \; ,$$

where

$$C = \sum_{i=1}^{n} \lambda^i = (\lambda - \lambda^{m+1})/(1-\lambda)$$

is a normalization factor. Square-Root replication has $r_i/R = \lambda^{i/2}/B$ where

$$B = \sum_{i=1}^{m} \lambda^{i/2} = (\lambda^{1/2} - \lambda^{(m+1)/2})/(1-\lambda^{1/2}) \; .$$

We thus have

$$\rho A_{optimal} = (B/C)\sum_{i=1}^{m} \lambda^{i/2} = B^2/C = \frac{1+\lambda^{1/2}}{\lambda^{1/2}+\lambda^{(m+1)/2}} \; .$$

Truncated Pareto Distribution $R_m(\alpha)$ is a truncation to m objects of a Pareto distribution with shape parameter α. Thus,

$$q_i = i^{-\alpha-1}/B_{\alpha+1,m} \; (i = 1, \dots, m) \; ,$$

where the normalization factor is

$$B_{y,m} = \sum_{i=1}^{m} i^{-y} \approx \int_1^m x^{-y} dx = \frac{m^{1-y}-1}{1-y} \; (y \neq 1) \text{ or } \ln m \; (y = 1) \; .$$

For sufficiently large m we can approximate

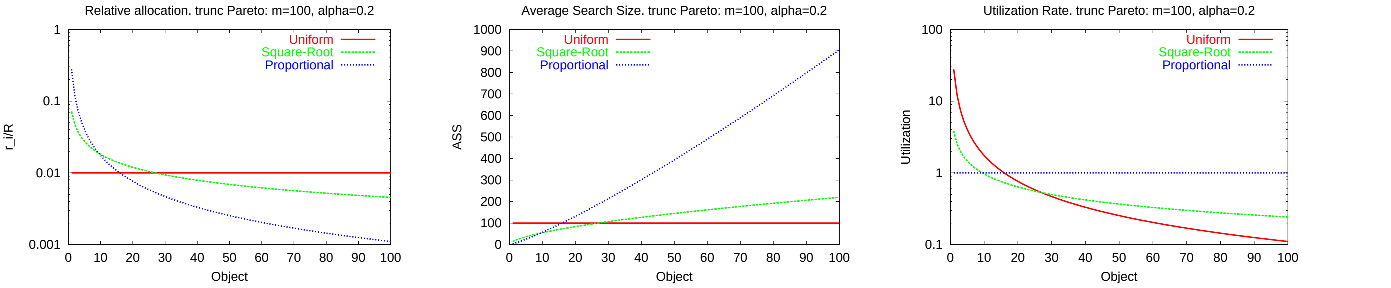
$$B_{y,m} \approx \begin{cases} \ln m & (y < 1) \\ m^{1-y}/(1-y) & (y = 1) \\ 1/(y-1) & (y > 1) \end{cases}$$

With Square-Root replication we obtain

$$r_i/R = i^{(-\alpha-1)/2}/B_{(\alpha+1)/2,m}$$

and average search size of

$$\rho A_{optimal} = (B_{(\alpha+1)/2,m})^2/B_{\alpha+1,m} \approx \begin{cases} 4\alpha m^{1-\alpha}/(1-\alpha)^2 & (\alpha < 1) \\ \ln^2 m & (\alpha = 1) \\ 4\alpha/(\alpha-1)^2 & (\alpha > 1) \end{cases}$$



Relative allocation r_i/R | Average search size ρA_i | Utilization rate U_i

Figure 5: Uniform, Proportional, and Square-Root strategies on a truncated Pareto distribution with $m = 100$ and $\alpha = 0.2$

We are now able to compute the *gain factor*, $A_{uniform}/A_{optimal}$, of using Square-Root rather than Uniform or Proportional replication. For Geometric distribution and for Pareto distribution with $\alpha > 1$, the optimal average search size is *constant*. Thus, the gain factor is $\Theta(m)$. The gain factor is $\Theta(m^{\alpha})$ for Pareto with $\alpha < 1$ and $\theta(m/\ln^2 m)$ for $\alpha = 1$.

Figure 5 helps visualize the different properties of the three replication strategies. Both Uniform and Square-Root allocate to popular objects less than their "fair share" and to less popular objects more than their "fair share" of replicas, but Square-Root does so to a lesser extent. The variance in average search sizes of different objects with Square-Root is considerably smaller than with Proportional. The maximum utilization rate with Square-Root, is much lower than with Uniform (although larger than Proportional which provides optimal load balancing). The same patterns occur for other values of m and α, but the gaps grow with the skew, thus, are larger for more objects (larger m) and larger values of the shape parameter value α.

5.2 Achieving Square-Root Replication

While the Uniform and Proportional strategies are significantly suboptimal with respect to the average query size metric, they both have the advantage of being easy to implement in a distributed fashion. Uniform replication merely calls for a fixed number of copies to be made for each object, and Proportional replication calls for a fixed number of copies to be made of the requested object after each query. The question is then whether we can achieve the optimal Square-Root replication strategy with a distributed algorithm.

Assume that each query keeps track of the search size, that is, how many probes it took before finding the object. Then let's assume that each time a query is finished, the object is copied to a number of sites proportional to the number of probes. This means that on average the i'th object will be replicated $\alpha\frac{n}{r_i}$ times each time a query is issued (where α is an arbitrary constant). Thus, the number of copies r_i can be roughly described by the differential equation

$$\dot{r_i} = q_i \alpha \frac{n}{r_i}$$

where $\dot{r_i}$ is the time derivative of r_i.

If we look at the ratio of two objects, ask how the logarithm of this quantity changes, we find that, setting $z_{i,j} = \ln \frac{r_i}{r_j}$,

$$\dot{z} = \alpha n \left(\frac{q_j}{r_j^2} - \frac{q_i}{r_i^2}\right)$$

Thus, Square-Root replication, $r_i = \lambda\sqrt{q_i}$ is a fixed point of this equation; once that allocation has been achieved, the ratios don't change (but the constant λ does as the total number of copies changes).

This heuristic calculation suggests that perhaps replicating proportional to the number of sites probed would yield Square-Root replication. In the next section we simulate a number of replication policies and evaluate their performance.

Our analysis above makes some implicit assumptions on the process governing the *deletion of replicas*. In particular, we mentioned two schemes for creation of new replicas: Proportional replication scheme where each query generates a fixed number of replicas and Square-Root replication scheme where each query generates number of replicas proportional to search size. The analysis of both schemes assumes that replicas disappear over time and new replicas are created. The steady state is achieved when the creation rate equals the deletion rate. For these schemes to achieve their respective fixed points, the lifetimes of replicas must be *independent* of object identity or query rate. Examples of deletion processes that have this independence are: assigning fixed lifetimes (or lifetimes from a fixed distribution) for each replica, subject replicas at each site to First In First out (FIFO) replacement, or perform random deletions. Interesting examples of deletion processes that *do not* have this independence property are usage-based replacement policies such as Least Recently Used (LRU) or Least Frequently Used (LFU). These policies could *impede* the Square-Root scheme: Recall that Square-Root replication has different utilization for replicas of different objects; thus, the scheme would have a different fixed point under LRU or LFU. Since the fixed-point of Proportional replication is such that all replicas have the same utilization rate, the fixed-point of that scheme is still LRU and LFU, but the variance in replica lifetime would increase and thus stability would decrease.

Note that unlike FreeNet's replication algorithm, the replication strategies studied here do not attempt to cluster certain group of objects in certain regions of the network. In other words, they do not produce any correlation between routing and object locations, or, "structure," in the P2P network.

6 Evaluation of Replication Methods

We observe that there are two replication strategies that are easily implementable. One is "owner replication", where, when a search is successful, the object is stored at the requester node only. The other is "path replication", where, when a search succeeds, the object is stored at all nodes along the path from the requester node to the provider node. Owner replication is used in systems such as Gnutella. Path replication is used in systems such as FreeNet.

The analysis in the previous section suggests that square-root replication distribution is needed to minimize the overall search traffic, and an object should be replicated at the number of nodes that is proportional to the number of search probes. If a P2P system uses the k-walker random walk as the search algorithm, then on average, the number of nodes between the requester node and the provider node is $1/k$ of the total nodes visited. Path replication in this system should result in square-root distribution.

However, an aspect of path replication that is not studied in the previous section is that it tends to replicate objects to nodes that are topologically along the same path. To understand how this impacts the overall search traffic, we also study a third replication algorithm, "random replication." In random replication, once a search succeeds, we count the number of nodes on the path between the requester and the provider, m, then randomly pick m of the nodes that the k walkers visited to replicate the object. "Random replication" is harder to implement, but the performance difference between it and path replication highlights the topological impact of path replication.

We design a set of dynamic simulations to study the three replication strategies: owner replication, path replication, and random replication. We look at how they perform in the Random graph network topology.

A simulation starts by placing the M distinct objects randomly into the network. Then the *Query Generator* starts to generate queries according to a Poisson process with average generating rate at 5 queries per second. The query distribution among the M objects follows Zipf-like distribution with a given α value. The α value for the results presented here is 1.20. (We also run simulations with $\alpha = 0.80$ and $\alpha = 2.40$. The results are similar.) For each query, a node (that doesn't have the requested object yet) is chosen randomly to start the query.

For the search method, we use the 32-walker random Walk with state keeping, with checking at every fourth step..

Each node can store at most *objAllow* objects (40 in our simulations). Every time a node wants to store a new object but its storage space is full, an object is randomly chosen to be tossed out (Random Deletion).

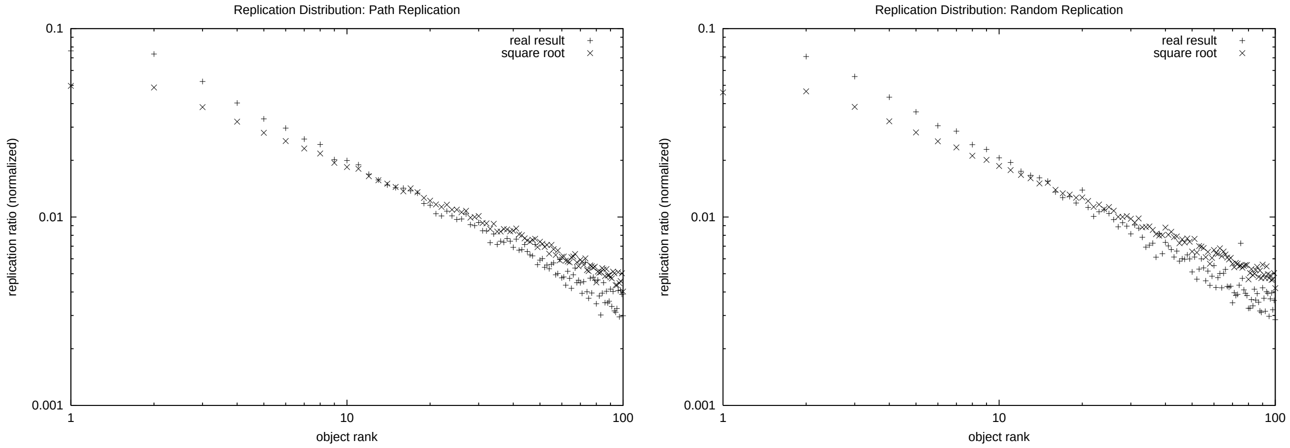


Figure 6: Distribution of replication ratios under Path replication and Random replication.

	Owner Replication	Path Replication	Random Replication
avg #msgs per node	56542.6	19155.5	14463.0
factor of improvement	1	2.95	3.91

Table 7: Message traffic of different replication strategies.

"Snapshots" are taken for every 2,000-query chunks. To allow for enough "warming up" process, we run each simulation for 10,000 seconds (which would generate about 50,000 queries given our generating rate), and look at the later part of the simulation.

For each replication strategy, we are interested in three questions:

- what kind of replication ratio distribution does the strategy generate?
- what is the average number of messages per node in a system using the strategy?
- what is the distribution of number of hops in a system using the strategy?

Figure 6 shows log-log plots of the distribution of replication ratios under path replication and random replication. We also plot the distribution that is the square root of the query distribution. Confirming our thereotical predictions, the results show clearly that both path replication and random replication generates replication ratios that are quite close to square-root of query ratios. (Due to space constraints we omit the graph for owner replication, except to say that it matches proportional distribution very closely.)

Table 7 lists the average number of messages a node has to process during the simulation. The result shows clearly the benefit of square-root distribution on reducing search traffic. Path replication and random replication reduces the overall message traffic by a factor of three to four. Hence, proactive replication such as path or random replication can improve the scalability of P2P systems significantly.

Much of the traffic reduction comes from reducing the number of hops it takes to find an object. Figure 7 shows the cumulative hop distribution for all queries under the three replication strategies. Path replication and random replication clearly outperform the owner replication; for example, the percentage of queries that finish within four hops are 71% for owner replication, 8691

The results also show that random replication improves upon the path replication. Thus, the topological effects of replicating along the path do hurt performance somewhat. Hence, if the implementation is not overly complex, a P2P system should adopt random replication instead of path replication.

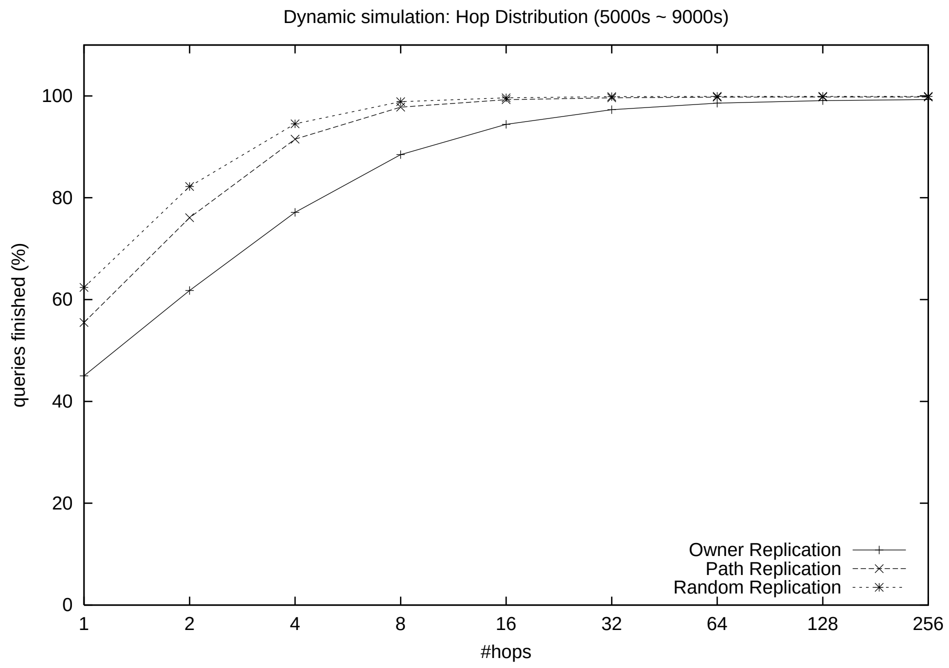


Figure 7: Cumulative distribution of the number of hops under the three replication strategies.

For P2P systems that do not want to store an object at nodes that have not requested it, are there ways to reduce the search traffic? We think the answer is yes. Such systems can still replicate the information that an object is stored at certain nodes following path or random replication, so that future searches for the object can be shortened. Each node can randomly delete a "hint" when it runs out of space to hold them all. As demonstrated in the above simulation, this simple step can improve the scalability of the system.

7 Related Work

As we mentioned in the Introduction, there are several different kinds of P2P networks. The highly structured P2P networks, such as CAN, Chord, Past, and Tapestry, all use precise placement algorithms to make searching efficient. However these systems have not been widely deployed, and their ability to operate with extremely unreliable nodes has not yet been demonstrated. Moreover, they cannot deal with partial-match queries (*e.g.*, searching for all objects whose titles contain two specific words).

There are also many loosely structured P2P networks. FreeNet [8], FreeHaven [10], MojoNation [14] are a but few examples of this rapidly growing list. Some of these systems, such as FreeHaven and MojoNation, focus on the trust, reputation management and security issues in Peer-to-Peer systems. Others, such as FreeNet, focus on file storage aspects of the system. Most of these loosely structured P2P networks use either directories or placement hints to improve the scalability of the search process. However, centralized directories don't scale well and placement hints don't handle partial-match queries.

Unstructured P2P systems like Gnutella can handle partial match queries, so the main question is whether their query performance can be made scalable, and that is what we focused on in this paper. We found that by adopting a k-random walk search method the performance of the search, in terms of load upon the network, could be improved by two orders of magnitude. Moreover, we found that the P2P network should not have a power-law degree distribution, nor resemble a mesh; random graphs P2P networks produce good results.

An interesting paper by Adamic *et al.* [1] studies random-walk search strategies in power-law networks, and finds that by modifying walkers to seek out high degree nodes the search performance can be greatly increased. However, such strategies greatly reduce the scalability of the search algorithm, which is our focus and not the subject of [1], because then almost all queries are sent to the very high degree nodes, making them bear the burden of almost the entire query load of the network.

The random walk search style is used in Freenet as well. There, the walk is guided; each node uses hints to help it choose which node to forward the query to. It also uses only one "walker". In comparison, our focus is on unstructured network, where hints are not available,

We found a wealth of information on Gnutella at web sites such as www.openP2P.com and gnutella.wego.com. We are also aware of a number of published research studies on the Gnutella network. For example, the freeloader phenomenon is examined in [2], and the topology and query distribution are studied in [19, 13]. However, none of these papers address the issue of better search algorithms or replication algorithms.

8 Conclusions and Future Work

This paper reports our simulation and modeling studies of several search algorithms and replication strategies for decentralized, unstructured peer-to-peer networks.

From simulation studies, we have learned that scalable search algorithm designs for such networks should consider three properties: adaptive termination, minimizing message duplication, and small granularity of coverage. The flooding algorithm being used In Gnutella does not satisfy any of the properties. We show that it generates a lot of network traffic and does not scale well. The expanding ring approach improves the flooding algorithm by using an adaptive termination mechanism. It can find data reasonably quickly while reducing the network traffic substantially, sometimes by an order of magnitude. The k-walker random walk with checking approach can find data more quickly while reducing the traffic further by another order of magnitude, because it reduces the granularity of coverage by using a fixed number of random walkers.

Our study on replication strategies show that for a fixed average number of replicas per node, square-root replication distribution is theoretically optimal in terms of minimizing the overall search traffic. Our simulations validated the theoretical analysis. We simulated owner, path and random replications, with the k-walker random walk with state keeping. Since path and random replications lead to square-root replication distribution, their overall message traffic is about four times less than the owner replication approach.

We have also learned from our simulation studies that uniformly random graphs are better for searching and data replication. The high degree nodes in power-law random graph and the current gnutella network bear much higher load than average and introduce more duplication overhead in searches. The results imply that it is better to form a uniformly random network topology using graph-building algorithms in peer-to-peer systems.

This study is our first step towards understanding the properties of scalable search algorithms, replication strategies, and network topologies for decentralized, unstructured peer-to-peer networks. There are still many open issues to study. It would be useful to model various search algorithms with certain network topologies and study them analytically. The k-walker random walk with checking and state keeping has a lot of rooms to improve. There is still a large gap between this algorithm and the optimal case in terms of minimum number of hops and minimum message traffic.

References

[1] L. A. Adamic, B. Humberman, R. Lukose, and A. Puniyani. Search in power law networks. In *In press, Phys. Rev. E, Vol. 64*, pages 46135–46143, 2001.

[2] Eytan Adar and Bernardo Huberman. Free riding on gnutella. In *First Monday, http://www.firstmonday.dk/issues/issue5_10/adar/index.html*, October 2000.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the www. In *Proceedings of 1996 International Conference on parallel and Distributed Information Systems (PDIS '96)*, 1996.

[4] Anonymous. reference removed for double blind reviewing. August 2001.

[5] Anonymous. reference removed for double blind reviewing. 2001.

[6] Clip2.com. The gnutella protocol specification v0.4. In *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*, 2000.

[7] Clip2.com. Gnutella: To the bandwidth barrier and beyond. In *Preprint, http://www.clip2.com/gnutella.html*, November 2000.

[8] Open Source Community. The free network project - rewiring the internet. In *http://freenet.sourceforge.net/*, 2001.

[9] Open Source Community. Gnutella. In *http://gnutella.wego.com/*, 2001.

[10] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability (LNCS 2009)*, July 2001.

[11] Dan Gallagher and Ronni Wilkerson. Network performance statistics for university of south carolina. In *http://eddie.csd.sc.edu/*, October 2001.

[12] Napster Inc. The napster homepage. In *http://www.napster.com/*, 2001.

[13] Mihajlo A. Jovanovic, Fred S. Annexstein, and Kenneth A. Berman. Scalability issues in large peer-to-peer networks - a case study of gnutella. Technical Report http://www.ececs.uc.edu/ mjovanov/Research/paper.html, University of Cincinnati, 2001.

[14] Jim McCoy. Mojo nation. In *http://www.mojonation.net/*, 2001.

[15] D. Plonka. Uw-madison napster traffic measurement. In *http://net.doit.wisc.edu/data/Napster*, March 2000.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, RichardKarp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM'2001*, August 2001.

[17] Jordan Ritter. Why gnutella can't scale. no, really. In *Preprint, http://www.darkridge.com/ jpr5/doc/gnutella.html*, 2001.

[18] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP'01*, 2001.

[19] Kunwadee Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. In *O'Reilly's www.openp2p.com*, February 2001.

[20] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM'2001*, August 2001.

[21] Megan Thomas and Ellen W. Zegura. Gt-itm: Georgia tech internetwork topology models. In *http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html*, 1997.

[22] Kelly Truelove. Gnutella: Alive, well, and changing fast. In *Preprint, http://www.openp2p.com/pub/a/p2p/2001/01/25/truelove0101.html*, January 2001.

[23] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Department, 2001.