Request Algorithms in Freenet-style Peer-to-Peer Systems

Jens Mache, Melanie Gilbert, Jason Guchereau, Jeff Lesh, Felix Ramli and Matthew Wilkinson
Lewis & Clark College
Portland, OR 97219, USA
{jmache, mgilbert, jwg, lesh, ramli, mjw}@lclark.edu

Abstract

In most peer-to-peer systems, edge resources self-organize into overlay networks. At the core of Freenet-style peer-to-peer systems are insert and request algorithms that dynamically change the overlay network and replicate files on demand.

We ran simulations to test how effective these algorithms are at improving the performance of subsequent queries. Our results show that for the original Freenet algorithms, performance improved less rapidly with a ratio of 99 requests to 1 insert than with an equal number of requests and inserts. This motivated us to design and test the performance of several new request algorithms. By changing the overlay network after failed requests and by further rewarding the fulfillers of successful requests, our new algorithms improved median pathlength by up to a factor of 9.25.

Keywords: peer-to-peer, overlay network, self-organization, request algorithm, performance evaluation

1 Introduction

As the Internet continues to experience rapid growth and ever increasing numbers of people request particular pieces of information, it gets exceedingly difficult to fulfill these requests (cf. Slashdot effect). Yet, to fulfill all the requests for one piece of information, there is an alternative to one powerful server (or server cluster): to rely on copies of the piece of information distributed across the network, so that many computers can fulfill these requests. This scalable and decentralized approach is an example of peer-to-peer networking. In most peer-to-peer systems, edge resources self-organize into overlay networks.

We are studying the performance of Freenet-style peer-to-peer systems in which every participating computer (node) automatically helps to fulfill requests by forwarding requests intelligently to one other node. At the core of Freenet-style peer-to-peer systems are insert and request algorithms that dynamically change the overlay network and replicate files on demand. We ran simulations to test how effective various request algorithms are in improving the performance of subsequent queries. Our results show that for the original Freenet algorithms, performance improved much slower with a ratio of 99 requests to 1 insert than with an equal number of requests and inserts. This motivated us to design several new request algorithms and test their performance.

The remainder of this paper is organized as follows: In Section 2, we provide background information and survey related work. In Section 3, we describe our experimental method. The original Freenet algorithms and their performance are discussed in Section 4. We present our new request algorithms and their performance in Section 5 and conclude in Section 6.

2 Background and related work

2.1 Freenet-style peer-to-peer systems

In Freenet-style systems [3, 4], every node helps to fulfill requests by forwarding queries (that it cannot fulfill itself) to the node in its routing table with the key closest to the key of the requested document. For the example in Fig. 1, node A forwards the request for the document with key 901 to node B because 222 is closer to 901 than 111 is.

Essential to good performance in a Freenet-style network is *specialization*, meaning that a node specializes in data within a dynamic and non-deterministic range. The network's insert and request algorithms are designed so that nodes specialize. However, specialization by itself is not enough. A node must also know about data that is different from its own area of concentration thereby reducing the pathlength of request queries. These *shortcut* connections allow requests to traverse areas of similar concentrations. Freenet's insert and request algorithms create shortcuts dynamically.

The occurrence of both shortcuts and specialization in a network results in short pathlengths, a property of small-

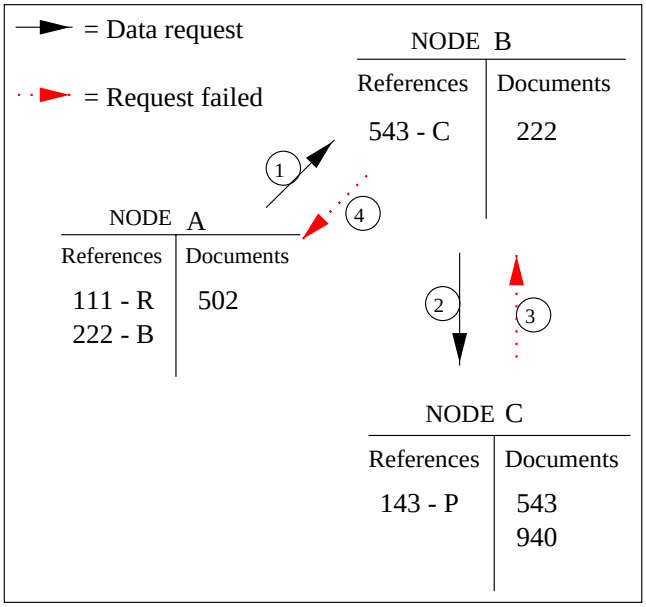

Figure 1. Failed request for key 901 (hops-to-live = 2).

world networks [15]. This term was coined when describing the social phenomena where most people are linked by short chains of acquaintances (popularly known as "six degrees of separation").

2.2 Related work

Freenet is very different from Napster [9] and Gnutella [5]. Napster's central broker was a single point of failure and could easily be attacked or shut down. Whereas Freenet operates depth-first, Gnutella operates breadth-first; for a Gnutella request, all neighboring nodes are contacted, which may in turn contact all their neighboring nodes. This can cause thousands of messages per request, making scalability a concern.

Similar to Freenet, Chord [13], CAN [10], Pastry [12] and Tapestry/Oceanstore [16, 7, 11] operate depth-first. However, these systems can be viewed as providing a distributed hash table, where nodes have fixed identities and data is placed deterministically. As a consequence, items can be located within a bounded number of routing hops. On the other hand, securing against attack, load balancing and exploiting proximity are more difficult.

Whereas modifications to the Gnutella and Pastry algorithms are described in [8] and [2], related work on modified algorithms for Freenet-style systems seems to be rare.

3 Experimental method

In this paper, our main concern is performance. Freenet-style peer-to-peer systems dynamically change the overlay network and replicate files on demand. We ran simulations to test how effective the request and insert algorithms are in improving the performance of subsequent queries (actions). Our measure of merit is *pathlength*. Short pathlength is important for (1) less overall bandwidth consumption, and (2) reduced probability of bad performance due to slow or unreliable links.

For our experiments, we use the Aurora simulator [1]. This simulator was designed to model the specifications of Freenet as described in [3]. We simulated a network of 1,000 nodes that are initially connected in a regular topology (four neighbors each). The network is evolved over a specified number of actions. *Actions* are either requests or inserts. They are interspersed randomly, given a request-to-insert ratio. Random keys are *inserted* at random nodes. Randomly-chosen existing keys are *requested* at random nodes. After every 100 actions, the performance of the overlay network is measured using a set of special probe requests (that do not modify the network).

Unless otherwise specified, *hops-to-live* (HTL) is 20 for inserts and 20 for requests. Each node has a datastore with a capacity of 50 documents and a routing table with up to 200 references. Results are averaged over 100 simulation runs.

4 Original algorithms and their performance

The standard request algorithm for Freenet can be described by the following pseudocode.

```
request(Key, Hops_to_Live, Passer)
  if (a document in my store matches Key)
    pass Document back to Passer
    return SUCCESS
  endif
  if (Hops_to_Live equals 0)
    return EXPIRED
  endif
  decrement Hops_to_Live
  while(references left in my routing table)
    in routing table find node with closest
     key that has not previously been tried
    request(Key, &Hops_to_Live, me)
    if(returned EXPIRED)
      return EXPIRED
    endif
    if(returned SUCCESS)
      add Document to my store //caching
      add reference pointing to Fulfiller
      pass Document back to Passer
      return SUCCESS
```

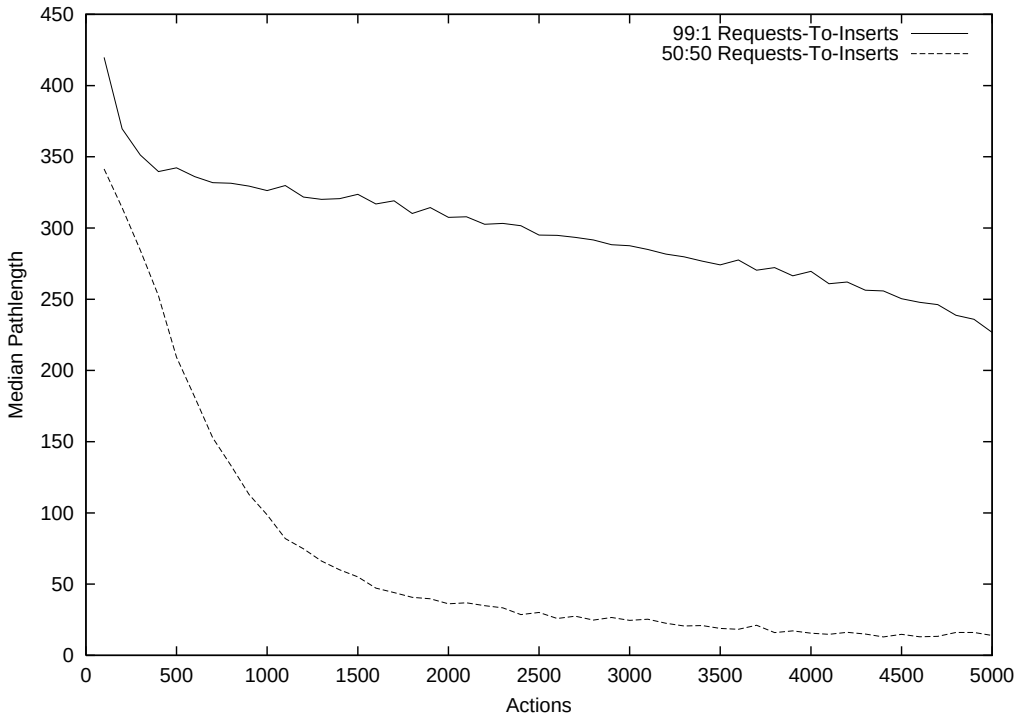

Figure 2. Pathlength decreases as the network gets used. However, the ratio of requests-to-inserts makes a big difference.

```
    endif
  endwhile
  return BACKTRACK
endrequest
```

The insert algorithm is very similar: each node passes the document on to the node in its routing table whose key is closest to the key of the document being inserted. Then, new references are added to the routing tables. While requests terminate upon finding a copy of the document, inserts typically fulfill their hops-to-live. An insert will only stop early when attempting to insert data to a node that already has it. This is called a collision.

As more requests are processed, the routing within the network becomes more efficient. As reported in [6], pathlength improves over time. However, our results show that pathlength improves much slower with a more realistic ratio of 99 requests to 1 insert than with an equal number of requests and inserts (see Fig. 2).

Looking closer at the simulation statistics, we notice that (1) many requests failed, and (2) successful requests added only a few shortcuts. For simulations involving 50% requests and 50% inserts (50:50 simulation) at 5,000 actions, 64% of request-queries failed. On average, successful request-queries are adding only about 4 shortcuts. This is in comparison to 20 shortcuts added by a non-collision insert.

These observations were our motivation to design and test several new request algorithms.

5 New algorithms and their performance

This section presents two new approaches. The first approach is improving *failed* (expired) request-queries by taking advantage of the work already done by the standard request process. The second approach is improving *successful* request-queries by adding extra references pointing to the fulfiller (using the remaining hops-to-live).

5.1 Learning from failed requests

Our goal was to take advantage of the work completed by failed request-queries. The last node in the request-query's path at which the hops-to-live expired is likely to be more specialized in keys close to the one being searched for than the original requesting node. Therefore, in our modified algorithm, the requesting node adds a reference to the last node's document that is closest to the one being searched for. For the example in Fig. 1, our algorithm adds "940 C" to node A's routing table.

This added reference turns out to be beneficial to pathlength. In the 99:1 (request-to-insert ratio) simulation, at 5,000 actions, the original Freenet algorithm has a pathlength of 229.33 (median) and 251.72 (average) to find a random existing document. The modified algorithm's average after 5,000 actions was down to 68.72 (median) and 110.48 (average) (see Fig. 3).

Why do these changes help? On a poorly connected network, the percentage of failed requests is initially very high, so this is

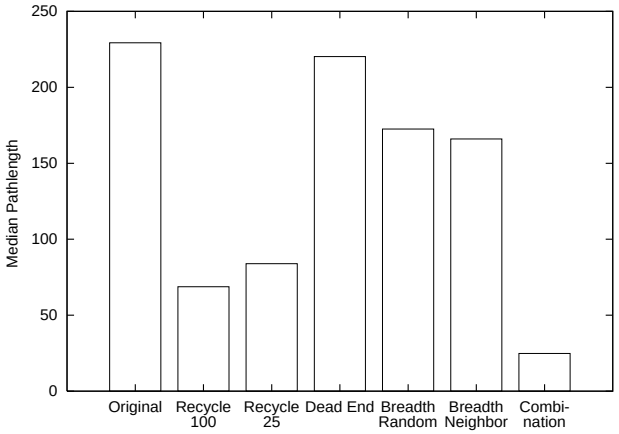

Figure 3. Pathlength for different algorithms (99:1 ratio, 5,000 actions)

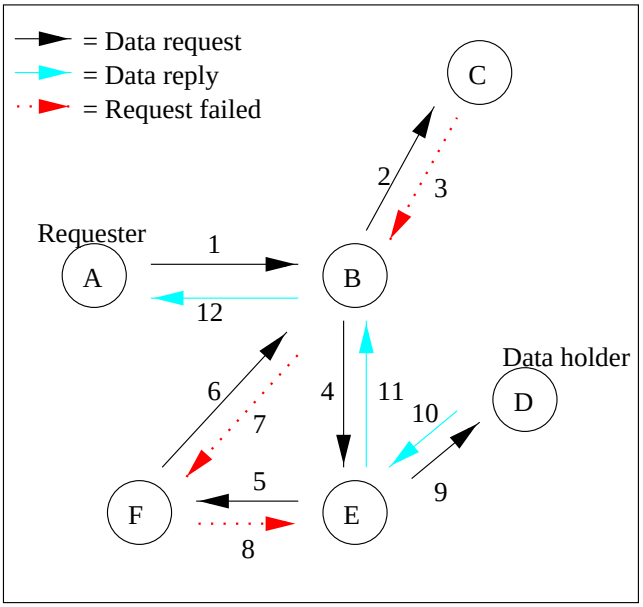

Figure 4. A request from node A eventually finds the data at Node D.

a crucial time to help the network learn more. The modifications to the request routine makes distant nodes more aware about the data that each other has, which is clearly beneficial for lowering request pathlengths.

An important consideration when changing any aspect of this intricate network is the effect your changes will have after long periods of usage. At 50,000 actions, our median pathlength was 6.27 compared to an original median of 2.94. The average pathlength was 19.69 compared to an original average of 17.35.

Since our new algorithm (which we call "recycle") did not perform as well in the long run, we tested a modified version that adds the new reference only 25% of the time. At 5,000 actions, the modified version had a median pathlength of 83.92 and average pathlength of 137.53, which both are a bit higher than previously. However, at 50,000 actions, the median pathlength of 2.88 was better than that of the original algorithm, and the average pathlength decreased to 7.08 (almost ten hops shorter compared to the original algorithm).

Another idea was to create a reference to the node along the path that had the closest key to the requested one. This was not as beneficial, as at 5,000 actions, the median pathlength was 91.28 and the average pathlength was 139.81.

5.2 Adding extra references on successful requests

Remember from Section 4 that requests stop upon finding a copy of the document and thus successful request-queries add only a small number of shortcuts. The second approach is improving *successful* request-queries by adding extra references pointing to the fulfiller (using the remaining hops-to-live). We describe three main strategies addressing how to add extra references: dead-end, depth and breadth. For ease of explanation, the explanations refer to Fig. 4, a typical request sequence in Freenet [3, 4].

5.2.1 Dead-ends

As a request-query traverses the Freenet network, it will sometimes reach a point where the query can go no farther. This may be caused by the node having little-to-no references in its routing table or by all the nodes that it can reference already having been queried about this request and thus will reject entering into a loop. As such, the node will pass the query back to the node from which it received the request, this is referred to as a backtrack and is exemplified in Fig. 4. Here, Node B receives a "request failed" from Node C. Node B then contacts its next-closest neighbor, and passes the query to that node.

However, the next-closest neighbor has a reference that is farther from the key requested than the node that the query backtracked away from. Thus, to improve specialization, if the information is found, it should be placed on the node that the query backtracked away from, which occurs in our "dead-end" algorithm. In Freenet, this could be done by the nodes remembering not only that they are part of the request query currently in operation, but also remembering who has backtracked to them. When the requested data is passed back through the path to the original requester, the nodes that remember being backtracked to would do an insert (hops-to-live = 1) to push the data onto the node that backtracked to it. In Fig. 4 this would result in Node C and Node F, who were originally thought to have similar information, to create references to the found data.

As seen in Fig. 3, the new "dead-end" algorithm performs only slightly better than the original algorithm. At 50,000 actions, our algorithm had a median pathlength of 2.8 connections, and average pathlength was 2.91.

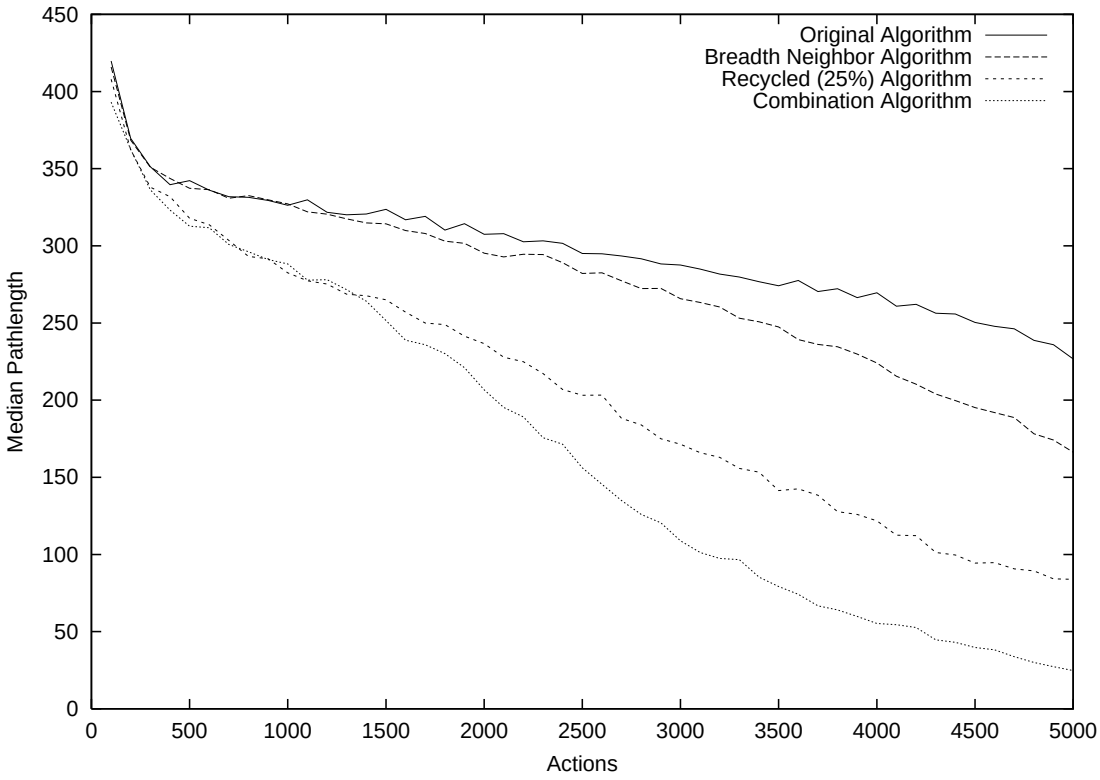

Figure 5. Median pathlength for different algorithms (99:1 ratio)

5.2.2 Depth

Another strategy to adding extra references is to start a normal insert at the fulfiller of a request-query (with the remaining hops-to-live). However, few shortcuts, if any, are added to the network due to collisions. Since each node that has the data knows where that data came from, the closest matching reference is the data's original source. As the original source typically has a copy of the data in storage, the insert algorithm identifies that a collision has occured, terminating the insert-query. Not surprisingly, the performance of this strategy and the original algorithm were very similar.

5.2.3 Breadth

Alternately, the fulfiller could "announce" the requested document (that it has fulfilled) to its neighbors in a breadth-first fashion. This means that the fulfiller only communicates with its immediate neighbors and that those neighbors do not pass any information on to their neighbors.

Here, our new algorithms look into the routing table of the fulfiller (node D in Fig. 4). The routing table contains a listing of references from Node D to a set of other nodes. For each remaining hop left, one additional node of the set adds a reference to Node D. We only consider nodes that do not already have the requested document or a reference to Node D for the requested key. This continues until either hops-to-live reaches zero or all references in the routing table of Node D have been checked.

Because Node D may have a very large number of references in its routing table, we must prioritize which ones are used first. We implemented two variations on this theme, plus have ideas for two more.

1) RANDOM. In our first attempt we chose random references in the routing table. Fig. 3 shows that this strategy produced better results than the original Freenet request algorithm.

2) NEIGHBOR. In the second variation, the greatest importance was placed on reinforcing specialization. The references chosen first were references from Node D that were closest to the key requested. This produced more favorable results than the random experiment described above.

3) FARTHEST. We are curious how doing the complete reverse of the previous variation will turn out, although we have not yet performed this experiment. Assuming this technique will decrease the specialization effect that is vital to Freenet, we predict that the outcome will not be favorable.

4) SMALL-WORLD MODEL. In this variation, the large majority of references is created among the closest neighbors of Node D's routing table. A small minority of references is added to more distant nodes. Again, we have not yet performed this experiment. We predict this variation to be the most effective at promoting clustering and specialization while making a few long-distant connections. This variation builds on the principles of small-world networks [15].

5.3 Combination

It was easy to combine the "recycle" algorithm (which modified the behavior of failed request-queries) with an algorithm that modified the behavior of successful request-queries. We chose "neighbor-breadth" due to its good performance.

This combination performed very well (see Fig. 5). The median pathlength at 5,000 actions was 24.79 and the average was 56.64, both are big improvements over our previously best algorithm, "recycle". Pathlength reached its minimum at approximately 10,000 actions, leveling off to a final average pathlength of 4.3.

6 Conclusions

At the core of Freenet-style peer-to-peer systems are insert and request algorithms that dynamically change the overlay network and replicate files on demand. We designed several new request algorithms and ran simulations to test their performance.

Our main conclusions are as follows:

- Freenet's original request algorithm is not very effective in changing the overlay network. This is due to (1) successful request adding only a small number of new connections and (2) failed request not changing the network at all.
- Our new "recycle" algorithm changes the overlay network on failed requests. It yields big improvements in the performance of subsequent queries, up to a factor of 3.34.
- We also tested several new strategies to further reward the fulfiller of successful requests. The "breadth-neighbor" strategy performed well, yielded improvements of up to a factor of 1.38.
- It is very beneficial to combine both ideas, "recycle" for failed requests and some form of reward for successful requests. Our simulation results showed performance improvements of up to a factor of 9.25.

Future work includes testing the variations described in 5.2.3 and additional long-term simulations.

Acknowledgements

We would like to thank the John S. Rogers Science Research Program and the Keck Foundation for their support.

References

[1] Aurora simulator. http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/freenet/aurora/.

[2] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2002.

[3] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, pages 40–49, January-February 2002.

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In H. Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66. Springer-Verlag, 2001.

[5] Gnutella. http://gnutella.wego.com.

[6] Theodore Hong. Performance. In Andy Oram, editor, *Peer-to-Peer – Harnessing the Power of Disruptive Technologies*, chapter 14, pages 203–241. O'Reilly, 2001.

[7] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Wstley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[8] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing (ICS)*, 2002.

[9] Napster. http://www.napster.com.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[11] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, pages 40–49, September–October 2001.

[12] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th International Conference on Distributed Systems Platforms (Middleware)*, 2001.

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

[14] Paul Thomas and Jens Mache. Evaluating the performance of peer-to-peer systems. In *Proceedings of the Oregon Academy of Science*, 2002.

[15] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

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