Look-Ahead Routing Reduces Wrong Turns in Freenet-Style Peer-to-Peer Systems

Jens Mache, Eric Anholt, Valentina Grigoreanu, Tim Likarish, Biljana Risteska
Lewis & Clark College
Portland, OR 97219, USA
{jmache, eta, vig, likarish, risteska}@lclark.edu

Abstract

Peer-to-Peer protocols and applications have drawn much attention. Freenet is a ground-breaking Peer-to-Peer system that protects the anonymity of information producers, consumers, and holders. However, it has been reported that Freenet has a "poor worst-case performance, because a few bad routing choices can throw a request completely off track" [9]. In this paper, we design and test look-ahead routing that reduces wrong turns and thus reduces the pathlength of data transfers. Each node checks with all of its immediate neighbors before continuing with the depth-first search. Results show a change in network traffic and a reduction in pathlength of up to 91% for 1-lookahead.

Keywords: peer-to-peer algorithm, Freenet, routing, performance evaluation

1. Introduction

The Internet continues to experience rapid growth and an ever-increasing number of requests for the same pieces of information and media. In a Peer-to-Peer (P2P) system, as information and media become more popular, they also become widely distributed and easier to access. In recent years, with the rise and fall of Napster [15] and replacements such as Gnutella [7] and Kazaa, P2P networking has drawn much attention. P2P networks attempt to allow efficient retrieval of data by having clients request data from each other, rather than connect to centralized servers.

In this paper, we focus on routing in Freenet-style Peer-to-Peer systems. The paper is organized as follows: In Section 2, we provide background information and survey related work. In Section 3, we analyze a case of poor performance due to wrong turns. In Section 4, we discuss algorithms that address wrong turns, and we present N-lookahead routing. Our simulation experiments and results are described in Section 5. After a discussion in Section 6, we conclude in Section 7.

2. Background and Related Work

2.1. Freenet-Style Peer-to-Peer Systems

Freenet [4, 5] is a groundbreaking Peer-to-Peer system that protects anonymity: no node in the network should be able to find either the inserter of new data or the original requester of a piece of data. Data files in the network are referenced by keys created by a hash function. Each node maintains a **datastore** that contains a cache of the most recently used files it has handled in the process of servicing requests and inserts. Each node also maintains a **reference table** (also called routing table) of keys it knows and nodes that might be able to handle a request for each of those keys.

"To increase network robustness and eliminate single points of failure, Freenet employs a completely decentralized architecture" [4]. Since Freenet does not use central servers to handle requests, each new node just needs to get initial references from other nodes. While the reference could be passed from one person to another out of band, it is typically done using lists of seed nodes that are either published on the web or sometimes fetched from Freenet using an existing node.

Freenet's routing algorithm is fairly simple. A Freenet request consists of a desired key and a hops-to-live (HTL) value, which is the maximum number of nodes to be contacted in the search for that key before the request expires. When a node receives a request, it first checks its local datastore for the entry. If the data exists in the datastore, the node returns the data and a reference to itself to the previous node, which would add the reference to its routing table and pass the data on, up the chain. If the data does not exist, it finds the **closest key** in its routing table to the requested key, and passes the request on to that node with a decremented HTL. The next node performs the same steps. If a node does not have the requested

data and cannot find a new node to route to, it returns failure to the previous node. The requester would then ask its next best choice with the HTL decremented. If a node detects that it is already part of a search, it returns failure, so that the network does not form loops. This process continues until we run out of HTL. At that point, a DataNotFound message will be returned up the chain. This can happen for one of two reasons: either the data was not found by taking the particular path that the query took, or the data simply does not exist in the network.

For the sake of anonymity, the algorithm above is not followed exactly. When a node is passing data up the chain, it may replace the reference to the supplier of the data with a reference to itself. This keeps nodes higher up the chain from knowing if the reference is to the node that originally contained the data or if it is to a node somewhere along the chain. Also, when a request with a HTL of 1 (a request only to the local node) is received, the node may randomly choose to forward it on. This, combined with the encryption of connections between all nodes, prevents an individual from probing its neighbors for incriminating data. This also guards from being able to prove which node actually had what data before the query happened. However, it also makes efficient routing difficult.

Because nodes have limited datastore capacity and replace files on a Least-Recently-Used (LRU) basis, when a node gets routed to for similarly-numbered keys and successfully supplies them, it will tend to keep those in cache and thus specialize on specific key values. This has been seen in the actual Freenet implementation [9].

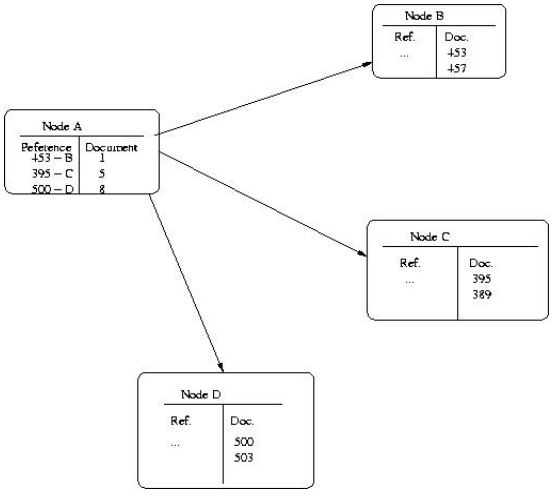

Figure 1: Sample reference table of a node "A".

Figure 1 shows a small example. Each node has a reference table and a datastore. Let us assume that a request is forwarded to node A for document 400. Node A checks its datastore, which does not contain the requested document. So it checks its reference table to see which of the keys is closest to 400. In our case, 395 is the closest key, so the query is then forwarded to node C. The reasoning behind this is that if all nodes follow this policy, nodes will end up getting specialized in certain keyspaces, as they get routed to for similar keys repeatedly and dissimilar keys tend to expire. If this is the case, it is beneficial for nodes to request from nodes with similar keys to what they are looking for; that node is more likely to have the requested key or be able to route to a node with the requested key, than a node with a reference for a more differing key value. The query continues in the same way until document 400 is found.

Essential features for the good performance of Freenet are the properties of a small-world network: specialization and building shortcuts [9]. A small-world network is a social phenomenon in which most people are linked by short chains of acquaintances. In the case of Freenet, each node will tend to specialize in a dynamic range of keyspace. It also builds bridges, or shortcuts, between itself and other nodes in the process of handling requests, using the node reference returned in successful queries.

The Freenet algorithm is surprisingly effective in using little bandwidth for searching, due to its serial nature. However, in order to retain Freenet's anonymity, data must be returned to the requester by passing the data back through the hop-by-hop chain instead of passing the data directly back to the requester. If the hop count (pathlength) is high, data retrieval can be quite slow. The longer the path, the higher the probability of slow or unreliable links on the path.

2.2. Related Work

Freenet is very different from Napster and Gnutella. 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 [20], CAN [16], Pastry [18] and Tapestry/Oceanstore [24, 10, 17] operate depth-first. However, these systems can be viewed as providing a distributed hashtable,

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 can be difficult.

Whereas modifications to the Gnutella and Pastry algorithms are described in [11] and [3], related work on modified algorithms for Freenet-style systems seems to be rare. Tests have been performed on modified datastore replacement schemes in one paper, but the request algorithm remained unchanged [23]. In our previous work, we designed and tested algorithms that help train Freenet-style systems by modifying the overlay network after failed request queries [13, 14] and after successful request queries [12]. However, we had not proposed a modified routing algorithm, until now.

Freenetproject.org released in July 2003 a specification for their Next Generation Routing Protocol which "is designed to make Freenet nodes much smarter about deciding where to route information" [6]. To improve routing, they place preference on faster nodes when routing queries occur. Three of the additional types of data that are collected about each node include: response times for requesting particular keys, the proportion of requests which succeeded, and the time required to establish a connection with that node. All of these factors are taken into account to give a "data reply estimation" time. Next Generation Routing thus decreases the chance of routing a query through a slow node and improves the adaptability of the network topology. However, no claim is made that this approach reduces the number of hops required to fulfill a query.

3. Poor Performance due to Wrong Turns

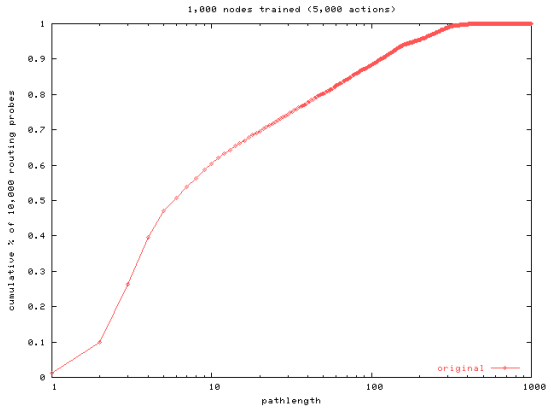

Figure 2: 1,000 node network trained for 5,000 actions with a median pathlength of about 5.

As reported by Hong [9], request pathlengths for random keys start out high in untrained network. For a 1,000-node network after 5,000 actions (inserts and requests) the median pathlength is below 10. This is an encouraging result. However, while the median is low, the average pathlength is around 35.4, see Figure 2. In this example, only 60.5% of requests are successful with an HTL of 10. Hence, 4 out of 10 queries are unfulfilled.

To evaluate Freenet's performance we ran a shortest-path algorithm (assuming global view) for the same network graph. In theory, the same requests could be fulfilled with an average pathlength of 2.17. See section 5.2 for more details.

From the tests, we observed that many requests get within 1 hop from the target key. However, the existing algorithm's heuristic causes the request to continue on for 200+ more hops than needed. Indeed, Hong writes that "Freenet has good average performance but poor worst-case performance, because a few bad routing choices can throw a request completely off track" [9].

Wrong turn in requesting document 14270 from:
Node 112

Reference	Node
*	*
3782	**91** (should follow)
*	*
14267	**173** (followed)
14372	324
*	*

Figure 3: Example of a wrong turn.

The example in Figure 3 outlines a request taken from a simulation of the original Freenet algorithm. Node 112 requests document 14270. Following the original Freenet algorithm described above, node 112 looks in its routing table for the closest document to 14270 and finds 14267 at node 173. Thus, the request is forwarded to node 173. In this example, the simulator follows this algorithm and finally finds the requested document 215 hops later. However, the pathlength could be 2, because node 112 has a reference to node 91 which has a copy of document 14270.

The original routing algorithm did not forward the query to node 91, because the routing table of node 112 only had a reference to node 91 for document 3782. One important emergent property of Freenet is that nodes specialize in the retrieval of some documents to the exclusion of others. This effect has been observed in actual Freenet nodes deployed in the Freenet network [6]. Figure 4 represents the keys stored by one node. The x-axis

represents the keyspace (from 0 to 2^160). The dark strips indicate areas in which the node has detailed knowledge about where requests for those keys should be routed. In our example, whereas node 91 has a copy of document 14270, node 112 knows only about node 91's knowledge about document 3782.

Figure 4: An actual Freenet node's stored keys. (Source [6])

4. Algorithms that Address Wrong Turns

4.1. Backtracking in Freenet

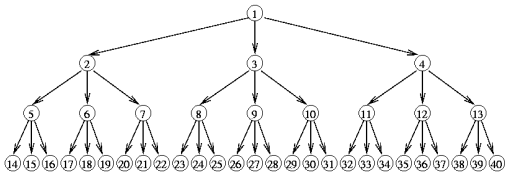

Figure 5: Sample graph of a network with 40 nodes.

The original Freenet algorithm tries the second choice if the first choice returns unsuccessfully and there is HTL remaining. This process is called backtracking.

Consider Figure 5 as a sample network graph. It is simplified because Freenet-style networks are not trees, nodes may have varying numbers of references to other nodes, and the nodes at depth 3 of this tree in Freenet would also have references, making backtracking unlikely.

Let's assume that the heuristic of "follow the reference to the closest key to that being requested" is represented by following the leftmost node on the tree. If we are looking for a document that only exists on node 7 and have an HTL of 10, the request will first go through the nodes in the order 1-2-5-14, and when 14 doesn't have the document it backs up to 5 which requests 15 and 16. When 5 has exhausted its references, it backs up to 2, which continues in the order 6-17-18-19. With 10 hops, backtracking has not reached node 7, and it has not examined the middle and rightmost subtrees.

4.2. Limited Discrepancy Search

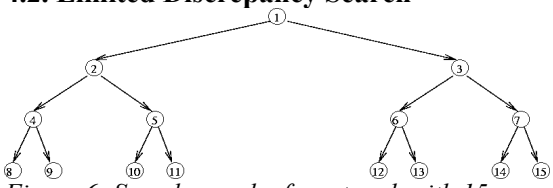

Figure 6: Sample graph of a network with 15 nodes.

Limited Discrepancy Search (LDS) [8] is an algorithm for backtracking when following the heuristic to its depth fails. It assumes that when a request fails initially (following the heuristic to its depth) the solution would most likely be found by taking a small number of deviations from the heuristic.

We will use the sample graph in Figure 6 fist to explain LDS, again with the closest-key heuristic being represented as the left branch. While the algorithm could be extended for graphs with out-degree > 2, the basic LDS algorithm assumes a binary tree.

In the LDS algorithm the first attempt is with 0 discrepancies from the heuristic, in the order 1-2-4-8. When 8 returns failure, the request backs up to 1, which then begins a 1-discrepancy search. Discrepancies (right branches) are used up first, so the search would go to the right at node 1 and continue in the order 3-6-12, for a total of 7 hops. If the document is not there, LDS would backtrack up to 1, then taken the order left-right-left, and if that failed left-left-right.

4.3. New N-Lookahead Routing

In our new N-lookahead algorithm, each node first checks with all of its immediate neighbors that are at most N hops away. If the data is not found, the depth-first search is continued. Fig. 7 shows an example of 1-lookahead. Compared to Gnutella, our "flooding" is of much smaller scale.

For another example, refer back to Figure 5. Again, we are looking for a document at node 7 with HTL=10 and the closest-reference heuristic is represented by taking the leftmost branch. With a 1-lookahead (N=1) algorithm, node 1 first checks its neighbors 2, 3, 4, and fails. It then sends its HTL=9 request to node 2, which does a 1-lookahead to 5, 6, and then finds the document at node 7. The HTL used is counted as 2.

If we were to do a 2-lookahead algorithm, node 1 requests that node 2, 3 and 4 do a 1-lookahead search. Node 2 requests from its neighbors 5, 6, and 7, and finds the data on node 7 with an HTL of 2.

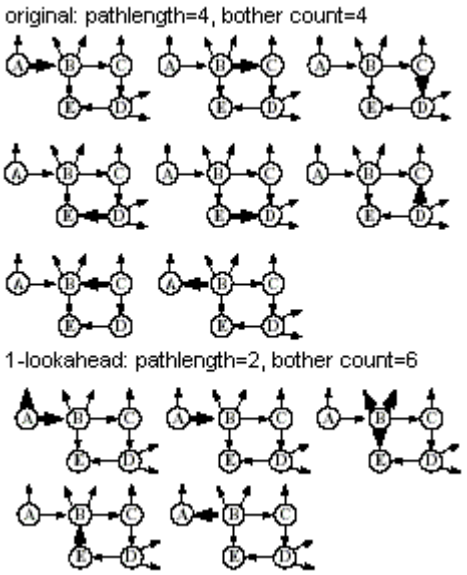

Figure 6: Example of original routing versus 1-lookahead routing (request originates at node A, data was at node E, node B did not know that).

Pseudocode for N-lookahead is as follows, modifications to the original algorithm are shown in **bold**:

```
node::receive_request(key, htl, is_lookahead){
if (key is in datastore)
  return SUCCESS; // File is transferred to req.
htl = htl – 1;
if (htl == 0)
  return EXPIRED;
ref = find_closest_reference(key) {
if (is_lookahead || htl > N) {
  if (is_lookahead)
    status=ref->receive_request(key, htl, TRUE);
  else
    status=ref->receive_request(key, N, TRUE);
  if (status == SUCCESS)
    return SUCCESS; // File is transferred to req
  ref = find_next_closest_reference(key, ref);
}
ref = find_closest_reference(key)
while (ref != NULL) {
  status = ref->receive_request(key, htl, FALSE);
  if (status == SUCCESS)
    return SUCCESS; // File is transferred to req
  if (status == EXPIRED)
    return EXPIRED;
  ref = find_next_closest_reference(key, ref);
}
}
```

5. Experiments

In this section, we discuss the assumptions made by Aurora and our methodology for testing the new algorithm.

5.1. Aurora Simulator

In order to evaluate our algorithm's performance, we used the Aurora simulator [1] for Freenet. Aurora is a publicly available simulator of the Freenet network, which is written in C++. Serapis [19] is its Java counterpart, which implements more of Freenet's complexity. Though it is useful for modeling Freenet, Aurora makes certain assumptions:

- A homogeneous network.
- All nodes are online at all times.
- Nodes all have an equally-sized datastore.
- Nodes all have an equal maximum number of references in their routing tables.
- Any node is equally likely to insert or request data as any other.
- Any data file is equally likely to be requested or inserted.
- Inserts and requests happen in approximately a 1 to 1 ratio.
- All data files are equally sized.

Aurora also doesn't account for bandwidth usage or differences in quality of network connections.

5.2. Test Cases

In order to test our algorithms, we first had to generate networks. We used the "train" target for Aurora, which generates a network of a certain number of nodes arranged in a regular graph (each node is connected to 2 neighbors each to its left and right) and then performs a number of iterations of either requests or inserts using the original Freenet algorithm. While we performed tests on networks with both 1,000 and 10,000 nodes, we will only present data on 1,000 nodes due to concerns we had with the scalability of the training process, which will be discussed in Section 6. For our 1,000 nodes we trained 5 networks each to 1,000, 5,000, and 50,000 iterations. We will call these "poorly trained," "trained," and "well-trained," respectively. Each node has a datastore with a capacity of 50 documents, and a routing table with up to 200 references.

Because of the way in which the graphs are generated, we know that they are weighted and directed. We used a modified all-pairs shortest path algorithm to find the average optimal pathlength for a random request from a random node. As expected, the average pathlength from a node to any other node decreased with training. Bridges and shortcuts get formed and nodes start filling up their reference tables, which facilitates getting across the graph faster. After averaging the data for the 5 different examples, we found that the average pathlengths for 1,000, 5,000 and 50,000 iterations were 4.79, 2.17, and 1.79, respectively. Additional data collected about the networks shows that, by the time networks are "well-trained", the pathlength to any node from any other node is either 1 or 2. The networks might therefore have been over-trained by that point, considering the low total number of nodes in the network. For "trained" networks, the pathlengths are between 1 and 4.

5.3. Methodology

We used the Aurora simulator to test our new lookahead algorithm on the networks outlined in section 5.2.

Tests were performed with N = 0 (no lookahead, or the original algorithm) and with N = 1, N = 2, and N= 3 (1-, 2-, and 3-lookahead, respectively). To reduce the effects of the randomness of training on the results, we ran tests on five of each type of network.

The test performed a "probe" which involved requesting 10,000 random documents that exist in the network, from random nodes, with an HTL equal to the number of nodes in the system. This ensures that the requests will succeed and report data on the hops taken. These requests are performed in a "frozen" state, so that they don't train the routing tables further or change datastores.

We also added a new metric to Aurora's output to be collected in the process of doing probes, which we call "**bother count**". This count measures the number of nodes contacted in the process of a request query. It differs from hop count which is HTL at start minus HTL at fulfiller. Hop count equals pathlength from fulfiller to requester (unless there was backtracking). In contrast, bother count includes nodes contacted as a part of the lookahead algorithm and nodes contacted that were already in the search. Let us keep in mind that decreasing hop count is most important. The bandwidth needed for transferring a file may be hundreds or thousands of times larger than that for a request message.

5.4 Results

While the 1,000-iteration "poorly trained" networks had around 20 references in each node's routing table, and the 5,000-iteration "trained" networks had around 50 to 100, the 50,000-iteration "well-trained" networks had reached the point where almost all nodes had full routing tables and were expiring old references in an LRU manner.

Table 1 shows the mean results for the 5 tests on each of our networks with 0- to 3-lookahead. "Average hops" is the average number of hops (new nodes contacted) it took for the request to succeed. "Average bothered per request" is the average number of node contacts that occurred in a request.

In the "poorly trained" networks, we see a small improvement with 1-lookahead over the original algorithm, and significantly higher bother counts. As N increased, the hops taken improved continually, to a point of 87% reduction at N=3**.**

	algorithm	average hop count	average bother count
poorly trained	original	168.523	903.490
	1-lookahead	123.457	2436.500
	2-lookahead	61.957	3055.090
	3-lookahead	21.621	2956.660
trained	original	27.785	137.030
	1-lookahead	3.598	112.280
	2-lookahead	2.882	111.530
	3-lookahead	3.579	99.280
well trained	original	31.317	180.560
	1-lookahead	2.815	233.280
	2-lookahead	2.907	238.130
	3-lookahead	3.686	209.660

Table 1: Performance of different algorithms (average of 10,000 requests, 5 different networks in each training category).

The "trained" and "well-trained" networks show an 87% and 91% improvement in hop count respectively when implementing a 1-lookahead search compared to the original protocol. They also showed an 18% reduction and 29% increase in bother count per request. The 5,000-iteration network showed an additional 20% improvement in hop count by moving to 2-lookahead, while the 50,000-iteration "well-trained" network stayed approximately equal. Both networks showed increases when moving to 3-lookahead.

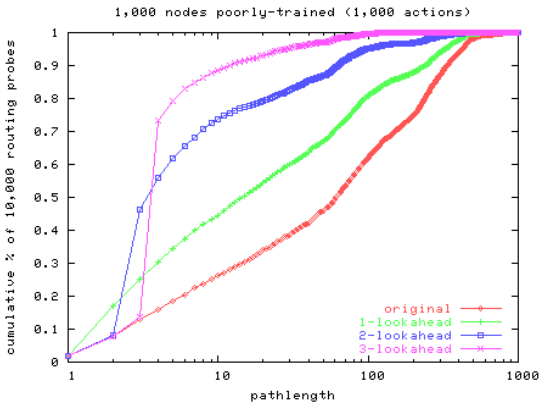

Figure 8: Distribution of hop count for different algorithms (1,000-iteration poorly- trained net).

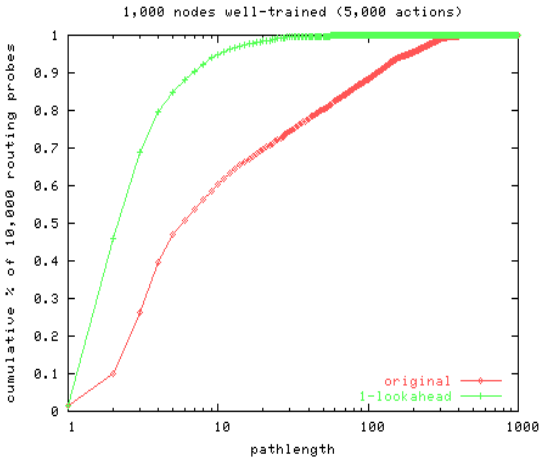

Figure 9: Distribution of hop count for different algorithms (5,000-iteration trained network).

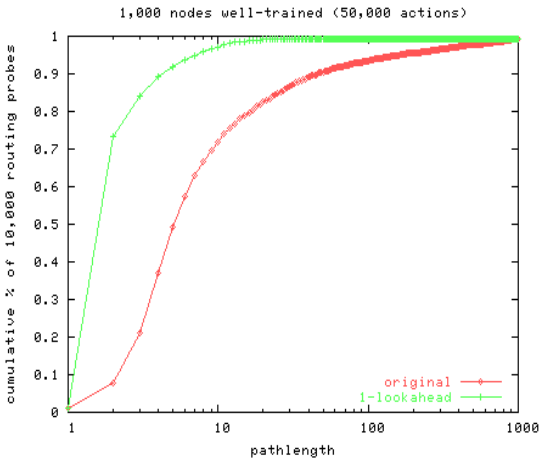

Figure 10: Distribution of hop count for different algorithms (50,000-iteration well-trained net).

The most important network to look at may be the 5,000-iteration "trained" network. At the point of 5,000 iterations the network is trained, but by 50,000 iterations each node is likely to have references to a large proportion of the total nodes, which is unlikely to be the case in a real network. Figure 9 shows that our 1-lookahead algorithm improves dramatically on the original algorithm. In 20 hops or less, 1-lookahead fulfills 98.4% of the requests, whereas the original algorithm fulfilled only 69.6%. To fulfill 90% of requests, it takes 7 hops, a huge difference from the 114 hops necessary with the original algorithm.

The reduction in node-bothering in the 5,000-iteration networks may come as a surprise. Intuition tells us that contacting every neighboring node in the lookahead search would increase the network traffic involved in searching, since the chance is small that a given neighbor would have the document. However, the results do make sense. As a request goes deeper in from its requester, a new node servicing the request is more likely to contact a node already involved in the search, because that node specialized in that keyspace. By reducing the number of hops, we also bother fewer nodes that are already involved in the search.

The increase in hops taken when changing from N=2 to N=3 in the 1,000-node 5,000-iteration test is also interesting. Obviously at this point the lookahead algorithm is suboptimal. Because the lookahead is depth-first instead of breadth-first, the 3-lookahead may be finding 3-hop routes to the data, that could have been fulfilled in 2- or 1-lookahead if the search had continued to nodes that appeared to be worse choices. However, a breadth-first search may not be useful to implement, because of the increase in network traffic. One alternative would be iterative deepening [22].

6. Discussion

While the 2-lookahead algorithm was more efficient in the 1,000-node, 5,000-iteration case, there are reasons to go with a 1-lookahead search. For an N=2 (or higher) lookahead algorithm to be implemented, Freenet would need to change their protocol and create a new type of "lookahead" request. This request would need to tell the other node in the search to contact all of its neighbors, instead. This differs from the current protocol where a node can only request that another node

performs an HTL-based search. The 1-lookahead algorithm, however, can be implemented using the current protocol because it only requires that a node sends a normal HTL=1 request to all of its neighbors, which will either return it from their datastores or fail (ignoring the anonymity features the contacted node may employ). The new N-lookahead protocol could also raise concerns that attackers could specify high-N requests to flood the network with traffic to a greater extent than they can with the standard HTL-based request.

Several things remain to be done. One of the most obvious is to conduct experiments to see if the effectiveness of the lookahead algorithm scales with network size and the size of routing tables.

We ran experiments on fifteen 10,000-node networks. In our 250,000 iteration tests, at 1 lookahead we found a 41% reduction in average hops taken, in exchange for a 970% increase in the bother count. Without lookahead, requests were taking on average 2367 hops, and even an improvement to 1376 is probably longer than any user would be willing to wait for a request if this were a real network.

While the results (with or without the new algorithm) don't appear to scale well with network size, we feel there are other variables involved that may be specific to the Aurora simulator. Because the training process begins with nodes in a regular graph, it is difficult to get started. At the beginning of training, the 20-HTL requests used will mostly fail because they can't reach across the graph to where a document is. The Aurora training simulator uses a 1:1 ratio of requests to inserts, which speeds up the process of training because the inserts will succeed and create many more node references in the network, which can be viewed as “shortcuts” between these nodes on the edges of the graph. The problem is that, with our 10,000-node network, many iterations are necessary to get the network trained so that connections are made between initially far-away nodes. By 250,000 iterations, there have been on average 125,000 inserts. With a datastore of 50 entries per node, each of those inserted documents would be in (10,000 * 50) / 125,000 = 4 nodes on average.

However, we estimate that, in the real Freenet network, the average user's datastore is many times larger than the data they insert. Their ratio of requests to inserts would also be very high – most people only read data, without publishing. Freenet nodes also don't join the network spontaneously in a regular graph; the network is already established, and most nodes will contact some seed nodes which are distributed through the network. Those seed nodes are also probably the most well-connected in the network, because of their increased popularity from new users connecting to them first. While we tried producing 10,000 node networks at with a 99:1 request:insert ratio and significantly expanded numbers of iterations, time constraints (due to the runtime of the simulation and probes) prevented us from completing those tests.

One of the significant possible problems with the lookahead algorithm is that it may have detrimental effects on the training of routing for the network. If successful requests pass through a sixth of the nodes they previously did, then a sixth of the routing information is being added to the network as a whole per request.

Another problem is the weakening of specialization. If a node is getting routed to for keys that are more randomly distributed, it won't tend to specialize as strongly in keyspace as it expires old data from its datastore.

More research will need to be done into how significant of a problem this reduction in routing information is. Some of our previous work on training improvements could be applied [12,13].

A possible way of counteracting the specialization problem mentioned above might be to use our lookahead algorithm more as the search nears the limit of its HTL (which, in a real network, would probably be near 20). However, if we are concerned with decreasing the mean HTL, wrong turns early on are the most expensive and thus the most important to perform lookahead on.

A possibility for limiting the bandwidth usage for requests might be to take more from the LDS idea. LDS suggests that improvement can be made by assuming that a heuristic is more likely to fail a small number of times. It may be that in our case a “wrong turn” that's avoided through lookahead is more likely to be in the first few routing choices a node has. If so, the lookahead could be limited to the first M best nodes in the routing table to improve both hops taken and bother count. More information would need to be gathered from our networks to see if this is true.

7. Conclusions

Wrong turns are detrimental to the performance of Freenet-style Peer-to-Peer systems. To reduce wrong turns, we designed look-ahead routing where a node first contacts its immediate neighbors before it continues the depth-first search. Our simulation results show that lookahead

routing improves the pathlength of data transfers. Our main conclusions are as follows:

- N-lookahead improves pathlength by preventing wrong turns when the data is within N hops of the node currently handling the request.
- 1-lookahead seems most promising. Average pathlength is reduced by up to 91%. Lookahead of N>1 can show additional reduction in pathlength over N=1, but has diminishing returns or may even increase pathlength on well-trained networks.
- Lookahead routing changes network traffic. Whereas the number of request messages sent can increase in poorly-trained networks, it is less significant or may even decrease in better-trained networks.

Future work includes the implementation of 1-lookahead routing in the reference Freenet node software. More experiments need to be conducted to see if the effectiveness of lookahead routing scales with network size and the size of routing tables. Lastly, because reduced pathlength results in fewer node references being added, it should be investigated how lookahead routing affects training. Some of our previous work on training improvements could be applied [12,13].

Acknowledgements

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

References

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

[2] Marcelo W. Barbosa, Melissa M. Costa, Jussara M. Almeida, V. A. F. Almeida, Using locality of reference to improve performance of peer-to-peer applications, In *Proceedings of the 4th Workshop on Software Performance (WOSP)*, 2004.

[3] 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.

[4] 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.

[5] 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.

[6] Free Network Project. Freenet's next generation routing protocol. http://www. freenetproject.org / index.php?page=ngrouting.

[7] Gnutella. www.gnutella.com

[8] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1995.

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

[10] 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.

[11] 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.

[12] Jens Mache, David Ely, Melanie Gilbert, Jason Gimba, Thierry Lopez, and Matthew Wilkinson. Modifying the overlay network of Freenet-style peer-topeer systems after successful request queries. In *Proceedings of the 37th Hawaii International Conference on System Sciences*, 2004.

[13] Jens Mache, Melanie Gilbert, Jason Guchereau, Jeff Lesh, Felix Ramli, and Matthew Wilkinson. Request algorithms in Freenet-style peer-to-peer systems. In *Proceedings of the 2nd IEEE International Conference on Peer-to-Peer Computing*, 2002.

[14] Jens Mache and Jeff Lesh. Simulated annealing and request algorithms in Freenet-style peer-to-peer systems. In *Proceedings of the 2003 International Conference on Internet Computing*, 2003.

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

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A

scalable content addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[17] 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.

[18] 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.

[19] Serapis. ttp://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/freenet/serapis.

[20] 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.

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

[22] Beverly Yang and Hector Garcia-Molina. Improving Search in Peer-to-Peer Networks, In *Proceedings of ICDCS*, 2002

[23] Hui Zhang, Ashish Goel, and Ramesh Govindan. Using the small-world model to improve Freenet performance. In *Proceedings of IEEE Infocom*, 2002.

[24] 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.