PathFinder: Efficient Lookups and Efficient Search in Peer-to-Peer Networks

Dirk Bradler, Lachezar Krumov, Max Mühlhäuser
TU Darmstadt, Germany
[bradler,krumov,max]@cs.tu-darmstadt.de

Jussi Kangasharju
Helsinki Institute for Information Technology, University of Helsinki, Finland
jussi.kangasharju@cs.helsinki.fi

***Abstract*—Peer-to-Peer networks are divided into two main classes: unstructured and structured. Overlays from the first class are better suited for exhaustive search, whereas those from the second class offer very efficient key-value lookups. In this paper we present a novel overlay, PathFinder, which combines the advantages of both classes within one single overlay for the first time. Our evaluation shows that PathFinder is comparable or even better in terms of lookup and complex query performance than existing peer-to-peer overlays and scales to hundreds of millions of nodes. Peers in PathFinder are arranged as Erdös Renyi random graph. Consequently, all overlay operations such as key-value lookup, complex queries and maintenance messages greatly benefit from the short average path length, the high number of alternative paths and the robustness of the underlying random graph topology.**

***Index Terms*—Peer-to-Peer, Overlay-Network, Key-Lookup, Exhaustive Search, DHT**

I. Introduction

Peer-to-peer overlay networks can be classified into *unstructured* and *structured* networks, depending on how they construct the overlay [1].

In an unstructured network the peers are free to choose their overlay neighbors and what they offer to the network.[1] In order to discover if a certain piece of information is available a peer must somehow search through the overlay. There are several implementations of such search algorithms. The original Napster used a central index server, Kazaa relied on a hybrid network with supernodes and the original Gnutella used a decentralized flooding of queries [1]. The BubbleStorm network [2] is a fully decentralized network based on random graphs and is able to provide efficient exhaustive search.

[1]In this paper we focus on networks where peers store and share content, e.g., files, database items, etc.

Structured networks, on the other hand, have strict rules about how the overlay is formed and where content should be placed within the network. Structured networks are also often called distributed hash tables (DHT) and the research world has seen several examples of DHTs [3], [4], [5], [6], [7], [8], [9]. DHTs are based on hashing peer and object identifiers and distributing the ID space among the peers. Typically, the number of messages needed to locate an object in a DHT grows logarithmically with the number of peers in the system. Thus, DHTs are very efficient for simple key-value lookups. Because objects are addressed with their unique names, searching in a DHT is hard to be made more efficient [10], [11], [12]. However, wildcard searching and complex queries either impose extensive complexity and costs in terms of additional messages or are not supported at all.

Given the attractive properties of both these different network structures: (i) human-friendly keyword searches in unstructured networks and (ii) computer-friendly and efficient lookups in DHTs, it is natural to ask the question: *Is it possible to combine these two properties in one single network?*

Our answer to the above question is PathFinder, a peer-to-peer overlay which combines an unstructured and a structured network in a single overlay. PathFinder is based on a random graph which gives it short average path length, large number of alternative paths for fault tolerable, highly robust and reliable overlay topology. Furthermore, the number of neighbors in PathFinder does not depend on the network size. Therefore, the load of individual peers in PathFinder remains constant even if the network grows up to 100 million or more peers. Our main contribution is the efficient combination of exhaustive searching and key-value lookups in a single overlay.

We evaluate PathFinder analytically and empirically and investigate its resistance to churn and its robustness. Our results clearly show that PathFinder is highly scalable, fast, robust and requires only a small per-peer state. In terms of exhaustive search performance PathFinder is comparable to BubbleStorm [2]. In terms of DHT-like lookup performance, our results show that PathFinder is at least as good as current DHTs and often requires even less overlay hops.

The rest of this paper is organized as follows. In Section II we present an overview of PathFinder. Section III compares it to existing P2P overlays. Sections IV and V discuss PathFinder's resistance to churn and attacks respectively. We conclude in Section VI.

II. PathFinder Design

In this section we present the system model and preliminaries of PathFinder. We also describe how basic processes like key-value lookup and exhaustive search work as well as how our overlay manages nodes joining/leaving the network and handles crashed nodes. Finally, we discuss how PathFinder can be built in a practical scenario.

A. Challenges

We designed PathFinder to be fully compliant with the concept of BubbleStorm [2], namely an overlay structure based on random graphs. We augment the basic random graph with a deterministic lookup mechanism (see Section II-D) to add efficient lookups into the exhaustive search provided by BubbleStorm. The challenge and one of the key contributions of this paper is developing a deterministic mechanism for exploiting these short paths in order to implement DHT-lookups.

PathFinder meets the following key-requirements:

- **Scalability:** Average path length of an object lookup grows with $ln(N)/ln(c)$, where N is the number of peers and c the average number of neighbors per peer.
- **Constant per-peer state:** The list of neighbors maintained by each peer does not depend on the network size.
- **Flexible exhaustive search:** Thanks to its underlying random graph topology, PathFinder supports exhaustive search with tunable success probability [2]. Any type of queries are supported.
- **Key lookup:** Locating an object in the network is competitive or even faster (in terms of overlay hops) than in other DHTs.

B. System Model and Preliminaries

All processes in PathFinder benefit from the properties of its underlying random graph and the routing scheme built on top of it.

Erdös-Rényi random graphs[2]*:* Random graphs have many attractive features, such as short average distance between the nodes and small diameter (both increase only logarithmically with the network size), high resistance against node failures, and the existence of several alternative paths between every two nodes in the network [13]. The average path length of a random graph can be estimated by $L = \frac{\log(N)}{\log(c)}$, where c is the average number of neighbors per node and N the number of nodes in the network. All these properties are highly desirable in any peer-to-peer overlay.

The challenge in building a peer-to-peer overlay on top of a random graph is that they have no structure, which implies that there is no rule stating which peer is a neighbor of which other peer. This is exactly the opposite of DHT overlays, which have construction principles allowing each node in the network to compute its neighbors in an unambiguous manner. This property enables DHTs to perform extremely efficient key lookups.

PathFinder's main contribution lies in defining a mechanism for reconstructing the neighbor list of another node in an Erdös-Rényi random graph. This gives us a very robust DHT network topology with straight-forward exhaustive search and exact key lookup mechanisms. Our solution allows for a *completely local reconstruction* of the neighbor lists; no additional network communication is required.

PathFinder construction principle: The basic idea of PathFinder is to build a robust network of virtual nodes on top of the physical peers (i.e. actual physical nodes). Routing among peers is carried out in the virtual network. The actual data transfer still takes place directly among the physical peers. PathFinder builds a random graph of virtual nodes and then distributes them among the actual peers. At least one virtual node is assigned to each peer. From the routing point of view, the data in the network is stored on the virtual nodes.

[2]In the rest of the paper we use the term random graph.

When a peer B is looking for a particular piece of information it has to find a path from one of its virtual nodes to the virtual node containing the requested data. Then B directly contacts the underlying peer A which is responsible for the targeted virtual node. B retrieves the requested data directly from A. This process is described in detail in Section II-D.

It is known that the degree sequence in a random graph is Poisson distributed. We need two pseudorandom number generators (PRNG) which initialized with the same ID always produce a deterministic sequence of numbers. Given a number c, the first generator returns Poisson distributed numbers with mean value c. The second PRNG given a node ID produces a deterministic sequence of numbers which we use as IDs for the neighbors of the given node.

The construction principle of PathFinder is as follows. First we fix a number c (see Section II-H how to chose c according to the number of peers and how to adapt it once the network becomes too small/large). Then, for each virtual node we determine the number of neighbors with the first number generator. The actual nodes IDs to which the current virtual node should be connected are chosen with the second number generator. The number generator is started with the ID of the virtual node. The process can be summarized in the following steps:

1) The underlying peer determines how many virtual nodes it should handle. See Section II-F for details.
2) For every virtual node handled by the peer:
   a) The peer uses the poisson number generator to determine the number of neighbors of the current virtual node.
   b) The peer then draws as many pseudo random numbers according to the number drawn in the previous step.
   c) The peer selects the virtual nodes with IDs matching to those numbers as neighbors for its current virtual node.

The following is a pseudo code implementation: The function `nextPoisson` is initialized with the current virtual node ID and returns a pseudorandom number from a Poisson distribution to determine the number of neighbors. The function `nextRandom` is initialized with the current virtual node ID as well and returns a deterministic random numbers uniformly distributed between 0 and N, where N is the number of virtual nodes in the network.

```
for each vNode.ID do
    numNeighbrs=nextPois(c,vNode.ID);
    ran_seed=init_ran_seed(vNode.ID);
    while  i < numNeighbors do
        neighID=random_seed.nextRan();
        vNode.store(neigh.ID);
        i = i + 1;
    end while
end for
```

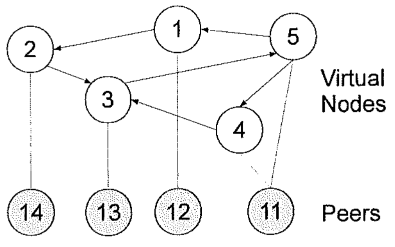
The construction mechanism of PathFinder allows the peers to build a random graph out of their virtual nodes. It is of crucial importance that a peer only needs a PRNG to perform that operation. There is no need for network communication. Similarly, *any peer* can determine the neighbors of *any virtual node*, by simply seeding the pseudo random number generator with the ID corresponding to the virtual node.

Now we have both, a random graph topology suited for exhaustive search and a mechanism for each node to compute the neighbor list of any other node. As we will discuss in detail in Section II-D, that is sufficient for any peer to contact any other targeted peer in the network by traversing just one single path, i.e. we can guarantee an efficient DHT behavior within the PathFinder overlay.

Note that neighbor links in the random graph are *directed*. The routing table of a peer is determined by the neighbors of its virtual nodes. It contains all the direct neighbors of all of its virtual nodes in the random graph. These tables are easy to maintain, because all peers hold only between one and two virtual nodes on average (i.e. c to $2c$ neighbors). As our results show, value of $c = 20$ is sufficient for good performance and better performance can be obtained for higher values of c. One entry in the routing table contains the virtual node ID and its IP address. Hence, the value of c could possibly be set much higher, routing tables with more than hundred entries are common in e.g. Kademlia, Pastry etc.

Routing table example of PathFinder: Figure 1 shows a small sample of PathFinder with a routing table for the peer with ID 11. The random graph has 5 virtual nodes (1 through 5) and there are 4 peers (with IDs from 11 through 14). Peer 11 handles two virtual nodes (4 and 5) and all the rest of the peers have 1 virtual node each. The arrows between the virtual nodes show the directed neighbor links.

Each peer keeps track of its own outgoing



Routing Table Peer 11

	Outgoing Links	Incoming Links
Node ID	3 1	3
Peer	13 12	13

Fig. 1. A small example of PathFinder.

links as well as incoming links from other virtual nodes. A peer learns the incoming links when the other peers attempt to connect to it. Keeping track of the incoming links is strictly speaking not necessary, but makes key lookups much more efficient (see Section II-D). The routing table of peer marked as 11 therefore consists of all outgoing links from its virtual nodes 4 and 5 and the incoming link from virtual node 3. In general, every peer is responsible for keeping only its outgoing links alive. In contrast to established DHTs, the maintenance costs of PathFinder does not depend on the network size as the average number of neighbors in the random graph is fixed.

C. Storing Objects

An object is stored on the virtual node (i.e. on the peer responsible for the virtual node) which matches the object's identifier. If the hash space is larger than the number of virtual nodes, then we map the object to the virtual node whose identifier matches the prefix of the object hash.

There is no need for an additional lookup service. When a peer is looking for an exact object, like a concrete file, it uses the hash function to compute the object identifier. Then the peer performs an efficient key lookup to the corresponding virtual node (DHT similar behavior). When the peer is looking for a range of objects, like all files containing a given regular expression in their titles, the peer performs exhaustive search which returns the object and its identifier (unstructured overlay similar behavior). Subsequent retrievals as well as parallel requests to replicas of the same object can be done by using the identifier to perform a lookup (Section II-D). This is a strong advantage of PathFinder as soon as one copy of a desired data object is found with exhaustive search, all remaining copies can easily be accessed using a subsequent key lookup.

D. Key Lookup

Key lookup is the process when a peer contacts another peer possessing a given data of interest. Using the structure of the network, the requesting peer traverses only one single and usually short path from itself to the target peer.

Key lookup is the main function of a DHT. In order to perform quick lookups, the average number of hops between peers as well as the variance needs to be kept small. We now show how PathFinder achieves efficient lookups and thus behaves as any other DHT. Suppose that peer A wants to retrieve an object O. Peer A determines that the virtual node w is responsible for object O by using the hash function described above. Now A has to route in the virtual network from one of its virtual nodes to w and directly retrieve O from the peer responsible for w.

Denote with V the set of virtual nodes managed by the peer A. For each virtual node in V, A calculates the neighbors of those nodes. (Note that this calculation is already done, since these neighbors are the entries in peer A's routing table.) A checks if any of those neighbors is the virtual node w. If yes, A contacts the underlying peer to retrieve O. If none of peer A's virtual node neighbors is responsible for O, A calculates the neighbors of all of its neighbors, i.e. its second neighbors. Because the neighbors of each virtual node are pre-known (see Section II-B), this is a simple local computation. Again, peer A checks if any of the new calculated neighbors is responsible for O. If yes, peer A sends its request to the virtual node whose neighbor is responsible for O. If still no match is found, peer A expands its search by calculating the neighbors of the nodes from the previous step and checks again. The process continues until a match is found. In worst case, A will have to calculate several neighbors, but a match is guaranteed.

For an average degree of c per virtual node, the above process requires us to compute c^i nodes for each step i. This becomes unwieldy for large networks which may require a large number of steps. For example, with $c = 20$ and 100 million nodes we need about 8 steps, i.e., $20^8 = 2.5 \cdot 10^{10}$ nodes. We mitigate this problem by *expanding the search rings from both A and w simultaneously*, as shown in Figure 2.

Because peer A is able to compute w's neighboring virtual nodes, A can expand the search rings *locally* from both the source and target

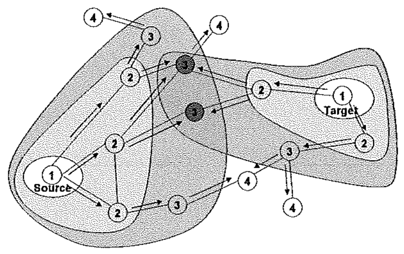


Fig. 2. Key lookup with local expanding ring search from source and target

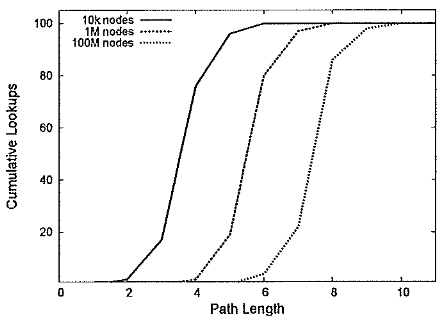


Fig. 3. Distribution of complete path length for 5000 key lookups with $c = 20$

sides, which is called forward and backward chaining. In every step the search depth of the source and target search ring is increased by one. In that way the number of rings around the source are divided between the source itself and the target. This leads to exponential decrease in the number of node IDs that have to be computed.

Now assume that the virtual node v is the intersection among the search rings around the source and the target. Recall that the edges among virtual nodes are directed, but that the underlying peers also keep track of their incoming links. That is, we now have a path from one of the virtual nodes of A to v and a path from w to v. All the nodes between w and v keep track of their incoming links. Therefore, they can also traverse the path backwards from v to w and thus provide A with a routing path to w.

The discovered path is passed along with the lookup message. Thus, every peer on the path knows immediately to which of its neighbors it should forward the query. In essence, PathFinder uses source routing for key lookups. Note that the whole computation of the path happens **locally** on the source peer. No additional messages have to be communicated. All costs come in the form of memory usage and computation time on the source peer. Those are however negligible for any regular computer: around 3.2 Megabytes of memory storage and simple integer computations on hashtables with several thousand entries. That was the maximum computer power required for carrying out the experiments described below.

We generated various PathFinder networks from 10^3 up to 10^8 nodes with average degree 20. In all of them we performed 5000 arbitrary key lookups. It turned out that, expanding rings of depth 3 or 4 (i.e., path length between 6 and 8) is sufficient for a successful key lookup, as shown in Figure 3. In the figure the x-axis shows the path length and the y-axis shows the cumulative fraction of observed paths. For example, for 1 million nodes the average path length is concentrated around 6. The theoretical average shortest path length for a random graph with 1 million nodes and average degree 20 is 4.6. The slight difference is caused by the forward/backward chaining.

Figure 3 also shows that increasing the network size by a factor of 100 leads only to two additional hops for key lookups. The key lookup performance depends mainly on the average number of neighbors c and only slightly on the number of virtual nodes N. It has been shown that the average path length scales with $O(\ln(N)/\ln(c))$ [13].

E. Searching with Complex Queries

PathFinder supports searching with complex queries with tunable success rate almost identical to BubbleStorm [2]. In fact, since both PathFinder and BubbleStorm are based on random graphs, we implemented the search mechanism of BubbleStorm directly into PathFinder. In BubbleStorm both data and queries are sent to some number of nodes, where the exact number of messages depends on how we set the probability of finding the match. We use exactly the same algorithm in PathFinder for searching and the reader is referred to [2] for details. The only difference is that the success probability in PathFinder is known in advance because it depends on the size of the random graph. The size of the random graph of virtual nodes is known.

F. Node Join and Leave

We now describe how peers join and leave the PathFinder overlay. The following invariants must hold at any time:

- All virtual nodes must be assigned to peers.

- Outgoing links must be maintained by the responsible peer.
- Incoming links are kept in the peer's state.

The purpose of the node join process is to integrate a new peer into the PathFinder overlay. The join process can also be used to automatically rebalance the network load, because virtual nodes may be unevenly distributed among the peers. When a new peer B wants to join the network it contacts a peer A that is already part of the network. The first virtual node assigned to B is calculated as a hash of its IP address. Peer A routes the join request using the key lookup procedure (Section II-D) to the virtual node which matches B's identifier. Let this node be handled by the peer C. C hands one or more of its virtual nodes over to B and informs the neighbors about the new peer B. In Section II-H we consider the case where C has no excess virtual nodes and B has to contact other peers to find a free virtual node and how the network adapts to such cases.

A successful join means that (i) a peer releases some of its virtual nodes to the new peer, but keeps at least one for itself and (ii) the new peer has successfully established connections to its neighbors. After the join process is completed, the new peer has at least one virtual node and an up-to-date neighbors table. The three invariants hold through the whole join procedure in the absence of node failures. We handle them shortly.

When a peer leaves the network properly, he/she hands all his/her virtual nodes over to his/her neighbors, which are then responsible for establishing connections to the underlying peers.

Observation: Peer joining/leaving the network causes on average $c + \ln(N)/\ln(c)$ messages.

The peer joining the network is routed to an arbitrary position determined by the hash function of its IP address. This costs one key lookup, which on average takes $\ln(N)/\ln(c)$ messages. The outgoing neighbors are directly transfered from the issuing node. Then, on average c incoming links transferred from the issuing node need to be updated, which causes additional c messages.

G. Node Crash

A node crash is the sudden departure of a peer from the network without correctly following the departure protocol from above. A crash violates the invariants from Section II-F and neighbors tables are no longer correct.

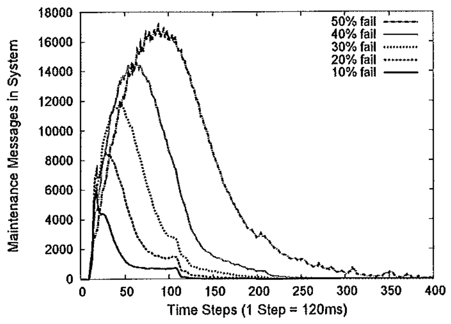


Fig. 4. Repair costs for network with 5,000 peers

The absence of the failed peer is recognized by its neighbors when they stop receiving keep-alive messages from it. The time for detecting a failed peer depends on the interval used for keep-alive messages. When a peer detects another failed peer it calculates locally all neighbors of its virtual node(s). The first peer in the sorted neighbor list has to take over the abandoned virtual node(s). Therefore the peer which has detected the failed peer sends the first peer in the list its own IP address and a message that it should start a recovery process. If the first peer in the neighbor list is not responding, it is replaced by the next peer in the list. Each peer which has detected the missing peer and still has not received a recovery message, also follows the above protocol. In case of concurrent attempts for taking over the abandoned virtual node, the peer with the smallest position in the neighbor list continues the recovery process.

After a responsible peer has been determined a new routing table for the failed virtual node has to be generated. For this purpose c key lookups are performed. After this step, which needs $c\frac{\ln(N)}{\ln(c)}$ hops on average, a routing table of all outgoing links of the abandoned virtual node is established.

The remaining incoming links are recovered automatically by the nodes pointing to the failed node. They notice timeouts of keep-alive pings sent to the failed node, because they still have the IP address of the crashed peer. They update their routing tables by performing a regular key lookup to the virtual node. This reveals the IP address of the new responsible peer in $c\frac{\ln(N)}{\ln(c)}$ steps on average.

Observation: A failed peer causes $2c(\ln(N)/\ln(c))$ messages on average to repair the network overlay.

Figure 4 shows the recovery messages volume for a simulated PathFinder network with 5,000 peers and different fractions of crashed peers. The x-axis shows the time in steps from the crash to full recovery. One step is at least one roundtrip time between two peers in real time. The y-axis shows the total amount of maintenance messages for each step in the whole system. The crash occurs at step 0 when the failed peers, 10–50% of all peers, disappear at once.

Moderate crashes (up to 30% crashed peers) are healed in about 100 simulated time units, and even a crash of half the peers is practically healed in 300 time units. The message load also remains reasonable, with about 36 messages per peer on average (half of 5,000 peers crashed in worst case and total number of messages in system is under 18,000). Considering the real-world time to heal the system, if 1 time unit is 120 milliseconds then the system would heal itself in 12 seconds for the smaller crashes and in 36 seconds for the 50% crash. The main determining factor is the ability of peers to send all the required messages in one step, so the recovery could potentially take longer. However, recovery times are still very short, on the order of a few minutes at maximum. Similar performance was observed as well in BubbleStorm [2].

H. Network Size Adaptation

Because virtual nodes can easily be transferred between peers, PathFinder is able to adapt itself to the current workload. Weaker peers may give up virtual nodes to stronger peers. To keep a reasonable ratio between peers and virtual nodes we have to keep track of the number of peers in the network. We estimate the number of peers in the PathFinder overlay with the push-sum gossiping protocol [14]. In [15] this procedure was extended to make it applicable for peer-to-peer networks.

The probability of finding a peer with more than one virtual nodes can be described through the hypergeometric distribution $f(1; N, m, c)$. A peer can ask any of its c neighbors for virtual nodes and success depends on the number of peers (m) with more than 1 virtual nodes within the network. When the ratio of peers in the network to average number of neighbors N/c is higher than 0.95, which is true in most of the cases, the process can also be approximated through the binomial distribution. A ratio of virtual nodes to peers of 1.15 establishes a successful join for 80% of all requests within 10 hops ($\mathcal{B}(10, 0.15)$) and over 96% after 20 hops.

We want to keep the cost for a join reasonable. Therefore, when the threshold of 1.15 virtual nodes per peer is reached, the network starts a transition phase in which the amount of virtual nodes is doubled. Assume that a virtual node w has just run the gossiping protocol and notices that the above threshold is reached. Then w starts the transition phase by generating two new virtual nodes, w_1 and w_2, by adding a new bit to the left of its own ID. The IDs of w_1 and w_2 are computed by attaching 0 respectively 1 to the old ID.

Now w_1 and w_2 have to calculate their neighbors. They proceed as in Section II-B, but use the new ID space. The IDs of the calculated neighbors also have one extra bit. Let x_1 be one of the neighbors w_1 has calculated. At this moment w_1 still does not have a routing table and cannot route to x_1. Therefore, w_1 disregards the left-most bit of x_1's ID and uses the routing table of w to determine the node x corresponding to this ID.

The peer responsible for x will be responsible for x_1 when the transition phase is over. This is because the ID of x_1 is the ID of x with one bit added on the lefthand side. Therefore, w_1 now adds this peer to its new routing table. If x has not yet started the transition phase, then it does so now. When the above procedure is carried for all new neighbors of w_1 and w_2, the transition phase for w is completed. When all virtual nodes have completed the transition, the old routing tables are abandoned.

Shrinking the ID space is also possible and works similarly. If the virtual node to peer ratio is higher than $4/1$, the amount of virtual nodes can be reduced in order to improve lookup performance. The procedure is the reverse of expanding the network, whereby peers will strip one bit from left of their IDs. In case two different peers map to the same shorter ID (quite likely), the one who had bit 0 as the first bit takes over. The peer who had bit 1 must find another virtual node to take over. If we set the threshold of shrinking high enough, over $4/1$, then there are enough virtual nodes for all peers, but a peer might have to search for a free one.

III. Comparison and Analysis

Most DHT overlays provide the same functionality, since they all support the common interface for key based routing. The main differences between various DHT implementations are average lookup path length, resilience to failures, and load balancing. In this section we compare PathFinder to other DHTs presented in the literature. We perform the comparisons both with simulations and analytically for networks which are too large to simulate (over 1 million nodes). All simulations were performed with the P2P simulator PlanetSim [16].

The lookup path length of Chord is well studied [17]. It is asymptotically: $L_{avg}(N) = \frac{1}{(1+d)\log(1+d)-d\log(d)} \log N$, where N is the number of peers in the network. The parameter d tunes the finger density. Usually Chord has finger density $d = 1$ and therefore $L_{avg} = \frac{\log(N)}{2}$. The maximum path length of Chord is $\frac{\log(N)}{\log(1+d)}$. The average path length of PathFinder is $\frac{\log(N)}{\log(c)}$, where c is the average number of neighbors. In other words, even for relatively small c, PathFinder has much shorter path length than Chord. The path length of the Pastry model can be estimated by $\lceil \log_{2^b}(N) \rceil$ [6], where b is a tunable parameter. The authors recommend $b = 4$. In this model, there are $\log_{2^b}(N)$ levels and $2^b - 1$ neighbors per level. This results in 96 neighbors for a network of 50 million peers. PathFinder achieves comparable results with only $c = 20$ neighbors on average. For $c = 50$ the average path length of PathFinder drops to 2/3 the path length of Pastry. Theoretically, PathFinder should achieve Pastry's performance for $c = 16$ (for $b = 4$). Since our results show that PathFinder matches Pastry already for $c = 20$, we suspect that Pastry's real-world performance for large networks would not be quite as good as the theoretical model let one expects. The Symphony overlay is based on a small world graph. This leads to key lookups in $O(\frac{\log^2(N)}{k})$ hops [18]. The variable k refers only to long distance links. The actual amount of neighbors is indeed much higher [18]. The diameter of CAN is $\frac{1}{2}dN^{\frac{1}{d}}$ with a degree for each node $2d$, with a fixed d. With large d the distribution of path length becomes gaussian, like Chord. The butterfly network has close to optimal diameter and average path length. The average distance in a butterfly network is given by [19]: $\mu_d \approx \frac{3\log_k(N)}{2}$. An implementation of the butterfly network, Viceroy [5], has an average path length of $3\log_2(N)$. The theoretical average path length of PathFinder is $L = \frac{\log(N)}{\log(c)}$. This is a property of its underlying random graph.

In summary, most well known DHTs and PathFinder have a path length scaling (up to a multiplicative factor) as $\log N$. In this sense, PathFinder performs similar, but it does have a

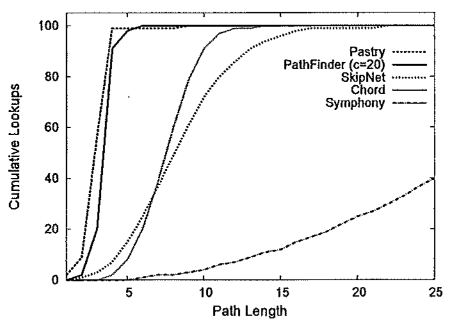


Fig. 5. Average number of hops for 5,000 key lookups in different DHTs

small and fix number of neighbors, independent from the actual network size.

We use simulations to evaluate the practical effects of the individual factors. We compare PathFinder with Pastry, Chord, Symphony, and SkipNet. Figure 5 shows the results for a 20,000 nodes network. We perform 5,000 lookups among random pairs of nodes and measure the number of hops each DHT takes to find the object.

Pastry and PathFinder have very similar performance, with the maximum number of hops being around 4. Chord and SkipNet perform worse, requiring on average 7 additional hops. Symphony's performance is extremely poor, with some lookups requiring up to 40 hops (not shown in the figure). CAN and Viceroy performed even worse and were dropped from further comparison.

We also perform an analytical comparison using the equations from the literature summarized above. Our goal is to gain some idea about how well the different networks scale to hundreds of millions of peers. We compare PathFinder with Pastry and Chord. We ignored Symphony because of its poor performance in the earlier experiment and SkipNet because of the lack of well-understood analytical model for its performance. We also used a DeBruijn-graph, because they are known to have optimal diameter.

Note that PathFinder results come from *actual simulation*, not analytical calculations. For the other overlays we have to resort to analytical modeling in order to estimate scalability for network sizes $> 10^6$ peers. Figure 6 displays the results. The x-axis shows the system size and the y-axis shows the average path length. As expected, Chord's performance is clearly poorer than that of Pastry and PathFinder. Pastry and PathFinder are very similar in performance for $c = 20$. Rising c to 50 gives PathFinder a similar to Pastry neighbors tables and yields about 1

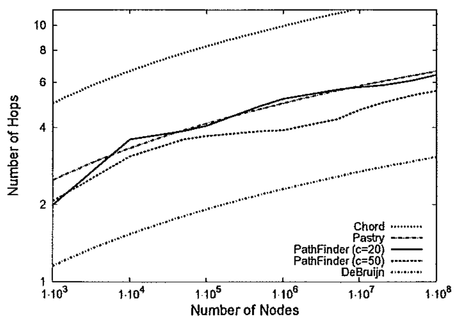


Fig. 6. Average number of hops for different DHTs measured analytically. Numbers for PathFinder are simulated.

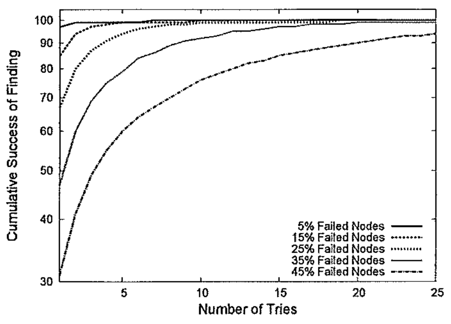


Fig. 7. Required lookup retries between each couple of nodes under churn in a network with 50,000 virtual nodes.

hop less in systems over 100 million nodes. The line for the DeBruijn graph shows the ultimative possible shortest path for the PathFinder network with c=20. PathFinder needs just ca. 1 hop more.

To summarize, with respect to average path length PathFinder performs very similar and at least as good as other known DHTs. In terms of scalability it benefits from the small and fix number of neighbors per peer and even network sizes of up to several millions of peers do perform well with 20 neighbors on average.

PathFinder also inherits the exhaustive search mechanism of BubleStorm. Hence, as an unstructured overlay it performs identical to BubleStorm and the reader is referred to [2] for thorough comparison to other unstructured systems.

IV. Resilience Against Failures

High churn rates [20] are common in peer-to-peer networks. Therefore, alternative paths may be needed to find a particular node. This is where PathFinder benefits from its random network topology. There are always as many alternative independent routes between any two nodes as the minimum of their degrees [13].

The challenge then is: How difficult is it to find a valid alternative path? Note that a peer A is not aware if there is a failed node on its path to peer B. It is first when A tries to reach B when A notices that it has to search for an alternative path. Due to the high number of alternative independent paths (paths which have only common start and end node) between each two nodes, the number of the required retries is very small.

We evaluated the performance of PathFinder under churn by generating a network of 50,000 virtual nodes and then consequently failing different fraction of them. Then we perform a key lookup using the procedure from Section II-D between each pair of remained nodes. For each pair we count the number of retries we have to make to get from the one node to the other. The results are shown in Figure 7. One observes that 5 retries are sufficient to connect over 90% of the remaining node pairs even when 25% of all nodes in the network have failed. In case that almost half of the nodes have failed, then 12 retries lead to successful lookup in 80% of the cases. In both cases the success rate is quite good. Note that in our tests we perform no maintenance in the overlay. After repairing all failed virtual nodes the number of retries drops back to zero.

Next, we address one crucial question remaining: How does the average path length change with the number of failed nodes?

In Figure 8 we plot the average shortest path length for $N = 50,000$ and different values of c. The x-axis shows the fraction of failed nodes and y-axis shows the average shortest path length. As one can see, the increases are minimal, even if half of the nodes suddenly disappear.

From Section II we know that key lookups not always follow the shortest paths. Therefore, we also evaluated the average lookup length under node failures of 25% and found that the maximum number of required hops for $N = 50,000$ and $c = 20$ increases merely from 6 to almost 7.

In short, the average path length and the number of required retries for key lookups in PathFinder stays stable even for severe fraction of failed nodes. Such a robust resilience is more than desirable in any peer-to-peer overlay.

V. Security and Other Issues

In terms of security, PathFinder faces the same challenges as most of the DHTs presented in the literature. Peers get their virtual node IDs as a hash of their IP address, which is the same as in other DHTs. Note that an attacker with access to a large pool of IP addresses may place herself/himself in a strategic position and discard

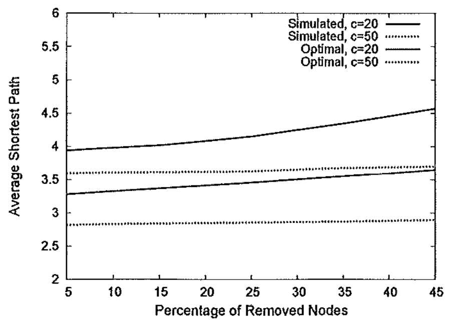


Fig. 8. Average shortest path length under churn.

or alter messages on their way to the destination. This is a common weakness of all DHTs.

If a malicious peer drops messages that are routed through it, the sending peer will eventually notice this because it does not get a reply. Recall from Section IV that there are as many node-disjoint paths between the sender and receiver as the minimum of their degrees. Thus, the sender can simply try again using a different path. The attacker is unlikely to be present on all the alternative paths. A simple approach to detecting malicious peers is to send a message always using two different paths. Given the short average path length of PathFinder a second or third parallel request does not impose high load on the overlay network, e.g. the P2P overlay network Kademlia sends parallel lookup requests by default. A comparison of the results will immediately suggest if the original message/reply have been modified.

Because PathFinder builds an overlay network with randomly selected neighbors, it is very likely that the virtual nodes do not match well with the underlying IP network. That is, first neighbors in the the overlay may be very remotely placed actual peers and messages among them have to travel significant distances. Other DHTs suffer from the same problem as well, since their routing algorithms also require contacting arbitrary peers in the network. DHT routing is typically optimized by selecting neighbors with small round trip times (RTT), with the goal of reducing the overall path latency. Similar approach works for PathFinder as well. When a peer needs to route a message in PathFinder it computes the shortest path as in Section II-D. If the peer notices that another of its neighbors has *considerably* shorter RTT than the "correct" peer, it can send the message to that neighbor. This may increase the path length in terms of hops as well as the required computational effort per peer, but might reduce the latency.

VI. Conclusions

In this paper we have presented PathFinder, an overlay which combines efficient exhaustive search and efficient key-value lookups in the same overlay. Combining these two mechanisms in the same overlay is very desirable, since it allows efficient and overhead-free implementation of natural usage patterns. PathFinder is the first overlay to combine exhaustive search and key-value lookups in an efficient manner.

Our results show that PathFinder has performance comparable or better to existing overlays. It scales easily to millions of nodes and its key lookup performance is in large networks better than in existing DHTs. Because PathFinder is based on a random graph, we are able to directly benefit from existing search mechanisms (e.g., BubbleStorm) for enabling efficient exhaustive search. We have shown the excellent performance and robustness of PathFinder both through simulations and by analytical means.

References

[1] R. Steinmetz, K. Wehrle, eds., *Peer-to-Peer Systems and Applications*, LNCS (Springer, 2005).

[2] W. Terpstra, J. Kangasharju, C. Leng, A. Buchmann, *Proc. SIGCOMM* pp. 49–60 (2007).

[3] M. Kaashoek, D. Karger, *IPTPS* pp. 98–107 (2003).

[4] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica, *IPTPS* **3** (2003).

[5] D. Malkhi, M. Naor, D. Ratajczak, *Proc. 21st symposium on Principles of distributed computing* (2002).

[6] A. Rowstron, P. Druschel, *IFIP/ACM* p. 329 (2001).

[7] B. Zaho, J. Kubiatowicz, A. Joseph, *Comp.* **74** (2001).

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, *Proc. SIGCOMM* pp. 161–172 (2001).

[9] C. Plaxton, *Theory of Comp. Systems* **32**, 241 (1999).

[10] Y. Yang, R. Dunlap, M. Rexroad, B. Cooper, *Proc. IEEE INFOCOM* (2006).

[11] J. Li, *et al.*, *IPTPS* (2003).

[12] P. Reynolds, A. Vahdat, *Middleware* (2003).

[13] B. Bollobás, ed., *Random Graphs* (Cambridge University Press, 2001).

[14] D. Kempe, A. Dobra, J. Gehrke, *44th IEEE Symposium on Foundations of Computer Science* p. 482 (2003).

[15] W. Terpstra, C. Leng, A. Buchmann, *26th ACM symposium on principles of distributed comp.* p. 390 (2007).

[16] P. Garcia, *et al.*, *Proc. 4th International Workshop, Software Engineering And Middleware* (2005).

[17] L. Zhuang, F. Zhou, Understanding Chord Performance, *Tech. Rep. CS268*, Department of Computer Science, UC Berkeley (2003).

[18] G. Manku, M. Bawa, P. Raghavan, *Proc. 4th USENIX Symposium on Internet Techn. and Systems* (2003).

[19] M. Hluchyj, M. Karol, *Lightwave Techn.* **9**, 1386 ('91).

[20] S. Saroiu, K. P. Gummadi, S. D. Gribble, *Multimedia Computing and Networking* (2002).