Implications of Neighbor Selection on DHT Overlays

Yingwu Zhu
Department of CSSE, Seattle University
zhuy@seattleu.edu

Xiaoyu Yang
Department of ECECS, University of Cincinnati
yangxu@ececs.uc.edu

Abstract

In this paper, we focus on how neighbor selection impacts DHT overlay structure, static resilience to failures and attacks, and lookup performance under churn. We present a deep comparison study of existing neighbor selection algorithms on the three aspects. We also propose a new neighbor selection policy and a new routing selection algorithm to improve lookup latency under churn. We hope this paper can provide some pieces of insight that will be useful in DHT routing-level designs, including neighbor selection and route selection.

1. Introduction

Motivated by widely-deployed file-sharing services, a large body of research work has focused on the design of structured peer-to-peer (P2P) systems, called distributed hash tables (DHTs) [10, 8]. DHTs map a large identifier space onto a set of nodes in a deterministic and distributed manner, and lookups are completed with $O(\log N)$ overlay hops in a network of N nodes where each node maintains $O(\log N)$ directional neighbor links.

While much previous work explores different applications of DHTs (e.g., distributed storage systems, application-level multicast and distributed web cache), our focus here is on the DHT's neighbor selection. Several neighbor selection policies, which are described in Section 3, have been proposed in this literature. Among them, proximity neighbor selection (PNS) [3] is a widely used technique to improve the performance of DHTs by choosing physically close nodes as routing table entries. Various neighbor selection policies (resulting from identifier constraints and different neighbor selection criteria) naturally raise a question: how do we make a choice? To answer this question, we need to understand the implications of neighbor selection. However, with few exceptions [1], most research work rarely evaluates and compares the impact of different neighbor selections on the *overlay structure*, *static resilience*, and *lookup performance under churn*.

In this paper, we attempt to take a small step in this direction by examining the impact of various neighbor selections on these three aspects. The first question we address is how different neighbor selections affect the DHT overlay structure. DHTs create a directional overlay graph where each node maintains $O(\log N)$ outgoing links. However, some neighbor selection policies may make some nodes more popular (e.g., they have high in-degrees) and thus may create an unbalanced overlay. We discuss this issue in Section 5.

The second question we explore is the static resilience of various neighbor selections to random node failures and targeted node attacks. The unbalanced overlay resulting from neighbor selection may affect the static resilience of DHTs. We examine this issue in Section 6.

The third issue we investigate is the lookup performance of various neighbor selections in the presence of *churn*. We discuss this issue in Section 7. Moreover, we show that long lifetime neighbor selection (LNS), a new neighbor selection we present in Section 4, performs much better under churn than current neighbor selection policies including PNS. With biased route selection (BRS), a new route selection algorithm (in Section 7), all neighbor selection policies can improve their lookup latency by up to 50ms under churn.

This paper explores the extent to which the neighbor selection impacts DHTs on the aforementioned three aspects. We hope it can provide some pieces of insight that will be useful in DHT routing-level designs (including neighbor selection and route selection). However, our paper itself has limitations. First, we only examine the impact of neighbor selection on *ring*-structured DHTs (e.g., Chord). The impacts on other DHT geometries such as *tree* and *hypercube* may need to be explored in our next step, though we speculate the conclusions for the ring-structured DHT may still hold for the other geometries. Second, LNS and BRS are based on the assumption that node lifetimes follow a Pareto distribution, though the assumption is consistent with the observations made by the measurement study of P2P systems [9].

2. Related Work

Our work is inspired by Gummadi et al. [3]. Gummadi et al. evaluated the impact of DHT geometries (e.g., ring, tree and hypercube) on the static resilience and proximity properties of DHTs. Our work differs from recent work [1] in the following aspects. First, we explore the lookup performance of various neighbor selections under *churn* rates as those observed in deployed P2P systems. Second, our proposed neighbor selection LNS considers node lifetime distribution to improve lookup performance under *churn*. Finally, we propose a new route selection algorithm BRS and explore its impact on lookup performance for various neighbor selections.

Perhaps one of the most studied aspects of DHT design has been PNS, a neighbor selection which makes neighbor choice by proximity. Dabek et al. [2] explored the impact of neighbor sample sizes (the number of neighbor candidates needs to be considered) on PNS. They showed the sample size of 16 achieves lookup performance close to the ideal. With few exceptions [7, 6], most previous work evaluated PNS in *static* networks. Bamboo [7] identifies and explores three factors that affect DHT performance under churn: reactive versus periodic failure recovery, message timeout calculation, and PNS. Li et al. [6] presented a performance vs. cost framework for evaluating and comparing various DHT protocols with different design choices under churn. Following the Pareto distribution of node lifetimes, Accordion [5] automatically adapts its routing table size to achieve low lookup latency in the presence of churn, while keeping bandwidth consumption within a user-specified budget.

3. Current Neighbor Selections

In the original set of DHT proposals, the neighbor choice for a node is extremely strict in the sense of identifier constraints. Deterministic neighbor selection (DNS) is such an example. By relaxing the rigidity in neighbor selection, some policies render a node some flexibility — that is, one can pick a neighbor according to some criteria (e.g., proximity) from a candidate pool which usually contains a few nodes satisfying certain logical identifier constraints (e.g., identifier range in Chord and prefix matching in Pastry). Among them, PNS and random neighbor selection (RNS) are such two examples. In what follows, we discuss DNS, PNS and RNS by taking Chord as an example DHT.

DNS. The neighbor choice for a node is purely deterministic and completely determined by the stringent logical identifier constraints. Given a set of node IDs in a DHT, the routing table entries for each node are fully determined. Consider a Chord node with ID a. The node maintains $O(\log N)$ neighbors [1](called *fingers*). DNS defines a specific set of fingers for the node: the i-th finger is the *first* node within the identifier range $[a + 2^i, a + 2^{i+1})$.

PNS. PNS improves DHT lookup performance by choosing physically close nodes as neighbors. For a Chord node with ID a, the i-th finger table entry is filled with a node which is physically closest to this node among all the nodes within the range$[a+2^i, a+2^{i+1})$. Note that the number of i-th finger node candidates could be as large as 2^i. An ideal PNS needs to search all these nodes for a nearby node. However, an exhaustive search consumes network resources, though it may improve lookup latency. Recent work [2, 3] proposed to use sampling as design tradeoff to choose neighbors — that is, PNS considers up to the first m nodes in the range $[a + 2^i, a + 2^{i+1})$ and picks the physically closest node among these m nodes as the i-th finger node. Dabek et al. [2] recommended the sample size m of 16. In the rest of this paper, we use 16 as the sample size in neighbor selection.

RNS. For a Chord node with ID a, RNS randomly chooses a node as the i-th finger node from those nodes in the range $[a + 2^i, a + 2^{i+1})$. Unless otherwise noted, RNS samples up to 16 consecutive nodes starting from the first node in the range $[a+2^i, a+2^{i+1})$ and randomly picks one of the sampled nodes as the i-th finger node.

4. Design of LNS

Current neighbor selection policies rely on the routing state *refreshing*/*maintenance* process performed periodically to maintain routing table freshness, in the face of churn as nodes continuously join or leave. The refreshing interval controls the degree of routing table freshness: short intervals can quickly detect and evict dead entries from routing tables, but at the cost of increased bandwidth cost. Moreover, the recovery process will compete with user lookups for bandwidth.

LNS bases its neighbor choice on node liveness information: each node attempts to populate its routing table with neighbors which tend to stay alive for a long time and evicts neighbors which are more likely to leave the system soon. LNS may mitigate the *tight* dependency of the routing table freshness on the refreshing interval, by having routing table filled with relatively stable nodes. In other words, LNS may still ensure liveness of its routing table entries with relatively long refreshing intervals, thereby reducing the bandwidth consumption by the recovery process as well as the bandwidth competition between the recovery process and user lookups. Moreover, the routing paths formed by relatively stable neighbor links may incur *less timeouts* for lookups due to node failures, thereby improving lookup performance under churn.

However, the key issue to LNS is how to determine whether a node will stay long or not in the network when

[1]Additionally, each Chord node maintains a list of successors (also called *sequential neighbors* in [3]) whose IDs immediately follow its ID clockwise on the ring.

making the choice of neighbors. LNS solves this issue due to the following factors: (1) The measurement study [9] of deployed P2P systems has shown that the distribution of node lifetimes is often heavy-tailed: nodes that have been alive for a long time tend to stay alive for an even longer time; (2) LNS only needs to be interested in the node liveness probability estimates rather than precise prediction of node lifetimes. In what follows, we discuss the design of LNS, starting with the distribution of node lifetimes.

4.1. Node Lifetime Distribution

Figure 1 plots the CDF of the measured Gnutella node lifetime distribution and the Pareto distribution. Note that the shape of the measured distribution closely matches that of the Pareto distribution. We thus assume the node lifetime follows a Pareto distribution in this paper.

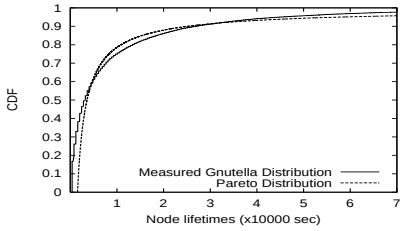


Figure 1. Cumulative distribution of the measured Gnutella node lifetime distribution [9] compared with a Pareto distribution with the shape parameter α= 0.83 and the scale parameter β= 1560 sec.

4.2. Node Liveness Probability

Given the Pareto distribution of node lifetimes, the probability of a node dying before time t is

$$P_{rob}(lifetime < t) = 1 - (\frac{\beta}{t})^{\alpha}$$

Let Δt_{alive} be the time for which a node has been staying alive, measured from the node's last join time to the time when it was last heard. Let Δt_{since} be the time between the time when the node was last heard and now (the current local time). The conditional probability of the node being alive, given that it has already been alive for Δt_{alive} seconds, is

$$\begin{aligned} p &= P_{rob}(lifetime > \Delta t_{alive} + \Delta t_{since} | \\ &\qquad lifetime > \Delta t_{alive}) \\ &= \frac{(\frac{\beta}{\Delta t_{alive}+\Delta t_{since}})^{\alpha}}{(\frac{\beta}{\Delta t_{alive}})^{\alpha}} = (\frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}})^{\alpha} \end{aligned} \quad (1)$$

4.3. Neighbor Choice

LNS allows nodes to use Equation 1 to calculate the liveness probability p for each neighbor candidates and choose the node with highest p as the routing table entry; if two neighbor candidates have same liveness probability p, LNS picks the one with lower latency [2]. In addition, if two neighbor candidates both have liveness probability p above certain threshold θ (in our experiments, we heuristically set $\theta = 0.9$), LNS picks the one with lower latency. Consider a Chord node with ID a. For the i-th finger node, LNS samples the first m nodes in the range $[a + 2^i, a + 2^{i+1})$ and chooses a neighbor according to the selection criteria.

As stated in Equation 1, calculating p requires three values: Δt_{alive} and Δt_{since} for a given node, and the shape parameter α of the Pareto distribution. Note that p does not depend on the scale parameter β. This is because β, determining the median node lifetime, is implicitly present in the values of Δt_{alive} and Δt_{since}. However, to calculate p, we still need to know the value of α, which is not easy to observe and calculate in real-world systems. To deal with this, we introduce a liveness probability indicator q as

$$q = \frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}} \quad (2)$$

Thus, we have $p = q^{\alpha}$, which is a monotonically increasing function of q. Big values of q mean high probabilities. In fact, LNS bases the choice of neighbors on q, the liveness probability indicator, rather than p. Next, we discuss how a node learns Δt_{alive} and Δt_{since} for each existing neighbor and neighbor candidates.

4.4. Learning Liveness Information

To calculate q for a node, we must know Δt_{alive} (the time between the node's last join and when it was last heard) and Δt_{since} (the time between now and when it was last heard). Thus, LNS requires each node keep track of its own Δt_{alive} and include this Δt_{alive} in every packet it sends. A node learns other nodes' liveness information (i.e., Δt_{alive}, Δt_{since}) as follows:

- When the node hears from an existing neighbor directly (e.g., by the maintenance traffic or lookup traffic), it records the current local timestamp as t_{last} in the corresponding routing table entry for that neighbor. The node also updates the associated Δt_{alive} with the newly-received Δt_{alive} value and resets the associated Δt_{since} to 0.

- When the node learns a "new" [3] node indirectly from another node (e.g., by the response for looking up neighbor candidates or neighbor replacements), it records the supplied Δt_{alive} and Δt_{since}, and sets t_{last} to the current local timestamp for this new node. This stored information will be used by LNS to make the neighbor choice or replacement if necessary.

Whenever a node needs to calculate another node's liveness probability indicator q, q can be computed as

$$q = \frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since} + (t_{now} - t_{last})} \quad (3)$$

[2] Note that LNS degenerates into PNS if we consider the same liveness probability for all the nodes in the system.

[3] The "new" node is new to the learning node, but it may not be new to the system.

where t_{now} is the current local timestamp. Whenever a node (say, A) informs another node (say, B) of node C's liveness information (e.g., the response for B's neighbor lookup request), A first calculates $\Delta t'_{since}$ by adding the locally stored Δt_{since} and the difference between t_{now} and t_{last}. Then, A sends to B the corresponding Δt_{alive} and the newly-calculated $\Delta t'_{since}$ as a substitute of Δt_{since}.

4.5. Discussion

LNS bases its neighbor choice on node liveness probability estimates. LNS aims to create more stable overlay links and minimizing the negative impact of churn. By creating relatively stable neighbor links, LNS attempts to avoid timeouts due to neighbor failures as much as possible to improve lookup latency under churn. Certainly, shorter routing state refreshing intervals may also be able to reduce timeouts for lookups to some extent, since the more frequently performed recovery process can quickly and timely detect and repair failures. However, shorter refreshing intervals have two main disadvantages: (1) With shorter intervals, the recovery process will consumes more bandwidth. (2) The more frequently performed recovery process will more intensely compete with user lookups for bandwidth, thereby impairing lookup performance to some extent. LNS, as will be shown in Section 7, does not require the recovery process perform too frequently, thereby reducing the bandwidth consumption by the DHT maintenance messages. Moreover, with relatively long refreshing intervals, LNS reduces the intense bandwidth competition between the recovery process and user lookups, and thus improves lookup latency.

5. Impact on Overlay Structure

In this section, we explore the impact of neighbor selection on the DHT overlay structure in terms of node out-degree and in-degree distributions.

5.1. Simulation Methodology

We use a discrete-event packet level simulator, called *p2psim* [6], to simulate Chord. P2psim does not simulate link transmission rate or queue delay. The finger table base of Chord is 2 used in our simulations. The number of successor nodes configured in our simulations are 4 or 16. The simulated network consists of $1,024$ nodes with inter-node latencies derived from measuring the pairwise latencies of $1,024$ DNS servers on the Internet using King method [4]. The average round-trip time for the simulated network is 152ms. Unless otherwise specified, our experimental results presented in this paper are based on this simulated network.

The simulations are initialized with one Chord node in the system. A new Chord node joins the system at a randomly-chosen time, until the total number of nodes reaches the bound (e.g., 1024 nodes). After system stabilization (i.e., each node has populated their routing tables), we calculate the distribution of node in-degree and out-degree for various neighbor selection policies. It is worth pointing out that the results below do not include those for LNS since the experiments are conducted in a static network (in the static network, each node has same liveness probability and LNS degenerates into PNS. Thus, the results for PNS can be regarded as those for LNS).

5.2. Experimental Results

Figure 2 plots the node degree distribution for various neighbor selection policies with 16 sequential neighbors. Due to space constraints, the results for 4 sequential neighbors are omitted but we observed similar characteristics. (In the rest of this paper, we use the terms *successors* and *sequential neighbors* [3] interchangeably. E.g., the *leaf set* nodes in Pastry are also called sequential neighbors.) Note that the node out-degree distribution is very balanced and same for all the neighbor selection policies. This is because each node maintains about $O(\log N)$ outgoing links, regardless of neighbor selection criteria. The slight difference among node out-degrees comes from the pseudo-randomness of the hash function used in Chord to generate node IDs; nodes are not truly randomly distributed on the Chord ring. Thus, some nodes may cannot find nodes to fill some of its finger table entries. The in-degree distribution for PNS and DNS is very skewed while the in-degree distribution for RNS is relatively balanced. The balanced in-degree distribution for RNS is due to its randomness in neighbor selection: each node within a range is equally chosen as other nodes' neighbors (though it may be affected by the sample size to some degree). This suggests that choosing certain metric (e.g., PNS) or stringent ID constraints (e.g., DNS) as neighbor selection criteria will produce an unbalanced overlay structure: some nodes are more popular and have higher in-degrees.

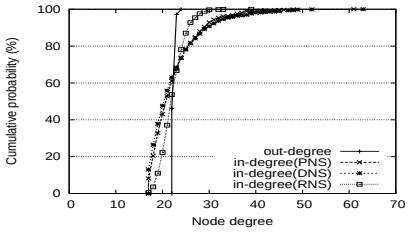


Figure 2. The distribution of node in-degrees and out-degrees. The number of successors is 16.

6. Impact on Static Resilience

Static resilience measures the DHT's tolerance to failures *without* the aid of recovery/maintenance mechanisms which periodically refresh routing table states and repair failures [3]. Static resilience indicates how quickly the recovery process has to perform, e.g., the stabilization (or

refreshing) interval. In the face of failures, a DHT with low static resilience must perform the recovery more frequently in order to maintain the same degree of robustness. In this section, we explore the impact of neighbor selection on static resilience in the presence of *random node failures* and *targeted node attacks*.

6.1. Simulation Methodology

The simulations are initialized with one Chord node in the system. A new Chord node joins the system at a randomly-chosen time, until the total number of nodes reaches the bound (e.g., 1024 nodes). After system stabilization (i.e., each node has populated their routing tables), we schedule random node failures or targeted node attacks.

In random node failures, we each time randomly pick a portion of nodes (0-90%) to fail. In targeted attacks, we each time remove a portion of nodes (0-90%) with highest in-degree in order to maximally weaken overall network reachability. When a portion of nodes fails due to either random failures or targeted attacks, the entries corresponding to the failed nodes are removed from the routing tables [4]. Then, each live node tries to route to every other live node. To measure static resilience, we use three metrics: (1) *% paths failed*: the percentage of routing failures between two live nodes. (2) *Mean successful lookup latency*: the average latency of successful routing between live nodes. (3) *Mean successful lookup hops*: the average overlay hops of successful routing between live nodes.

6.2. Experimental Results

Random node failures. Figure 3(a) shows the results for % failed paths as the % failed nodes is varying. The neighbor selection has not much impact on failed paths. This is because, each node in Chord maintains $O(\log N)$ outgoing links. Random node failures make each outgoing link to fail with similar probability, regardless of neighbor selection policies. Adding 16 sequential neighbors on each node significantly increases path failure resilience for all three policies. This suggests that DHTs, when equipped with more sequential neighbors, can be more tolerant to high node failure rates.

Figure 3(b) shows the results for average successful lookup latency as the % failed nodes is varying. Figure 3(c) shows the results for mean successful lookup hops. Several observations can be drawn: (1) As more nodes fail, the lookups take more hops and have higher latencies since the most preferable routing paths (approaching the destination in a greedy way in the absence of node failures) are likely to fail and lookups have to try alternate routing paths. However, when the percentage of failed nodes reaches beyond certain point (called *turning point*), those lookups involving more hops are more likely to fail. Therefore, the lookup hops (and thus latencies) drop after the turning point. (2) The addition of 16 sequential neighbors makes the turning point move from 65% to 80% failed nodes because many lookups will succeed by passing through the additional sequential neighbors, though at the cost of increased hops and latencies. (3) Among all these neighbor selections, PNS achieves the lowest latencies for lookups.

Targeted node attacks. Figure 4(a) shows the results for % failed paths as the % attacked nodes is varying. Two main observations can be drawn from this figure. First, with 4 sequential neighbors, RNS shows the best tolerance to attacks. This is because PNS and DNS produce a more skewed in-degree distribution among nodes; taking down the high in-degree nodes will make routing paths more vulnerable to attacks in PNS and DNS. Second, the addition of 16 sequential neighbors dramatically increases attack tolerance in all the three policies. Surprisingly, DNS and PNS show stronger resistance to attacks than RNS. The is because, when a large number of high in-degree nodes (e.g., $\geq 50\%$) are attacked, the relatively balanced in-degree distribution of RNS more likely causes partitioning the Chord ring, i.e., a number of consecutive nodes on the ring all are taken down due to their similar in-degrees [5] while this is less likely to happen in PNS or DNS due to their skewed in-degree distribution [6]. We thus believe that adding sufficient sequential neighbors will make the attacks on high in-degree nodes a lesser issue in the unbalanced overlay resulting from PNS or DNS.

Figure 4(b) shows the results for average successful lookup latency as the % attacked nodes is varying. Both lookup hops and latencies show similar characteristics to those in random node failures. Again, PNS shows the best lookup latency due to its neighbor selection metric. However, with 16 sequential neighbors, RNS show lower latencies than PNS when the percentage of attacked nodes is large (i.e., $\geq 50\%$). This is because RNS has higher lookup failure rate than PNS. Some failed lookups in RNS will succeed in PNS by going through the sequential neighbors, but at the cost of increased hops and latencies. Similar to PNS, DNS's path failure resilience comes at the cost of increased hops and latencies.

Impact of sequential neighbors. Figure 5 shows the results for PNS with respect to the number of sequential neighbors. Due to space constraints, we omit the results for other neighbor selection policies. But, we observed similar characteristics in them. Note that adding more sequential neighbors significantly increases path failure tolerance. However, this comes at the price of increased lookup latencies and hops (the results for hops are not shown here due

[4] Normally, these dead entries will be detected and repaired by the recovery process performed periodically in DHTs.

[5] In this case, even the sequential neighbors cannot bridge the partition.

[6] In PNS and DNS, the consecutive nodes on the Chord ring are likely to have different in-degrees due to the preference in neighbor choices.

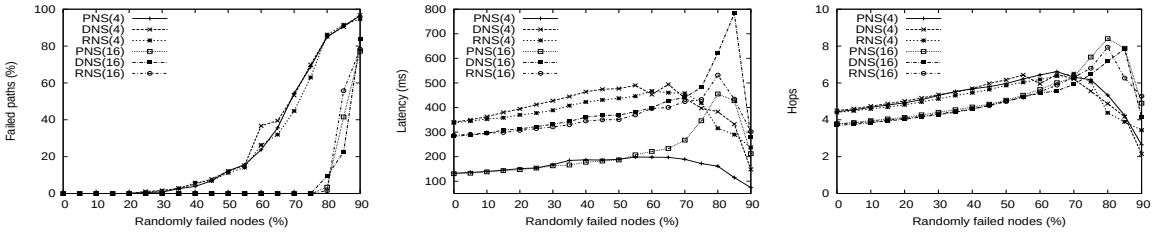


Figure 3. Results for varying percentage of node failures across different neighbor selections. The number in parenthesis is the number of successors. (a) Percentage of failed paths. (b) Average successful lookup latency. (c) Average successful lookup hops.

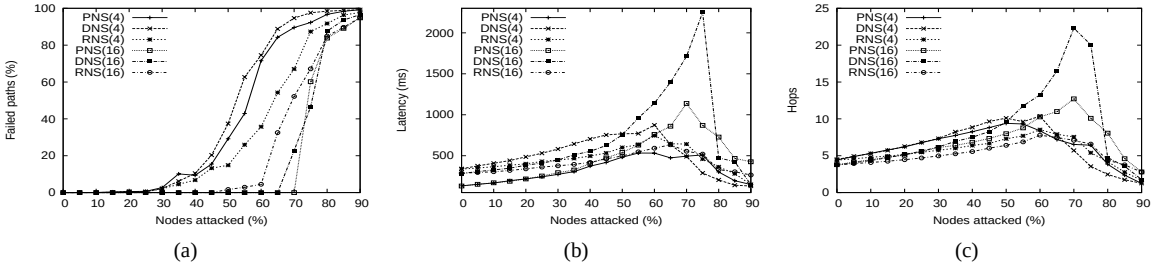


Figure 4. Results for varying percentage of attacked nodes across different neighbor selections. The number in parenthesis is the number of successors. (a) Percentage of failed paths. (b) Average successful lookup latency. (c) Average successful lookup hops.

to space limitations) as many lookups have to pass through the sequential neighbors to succeed.

Summary. Although the unbalanced overlay structure produced by PNS or DNS will make routing paths more vulnerable to attacks on high in-degree nodes, the addition of a *sufficient* number (e.g., 16) of sequential neighbors makes this a lesser issue. Surprisingly, with the aid of a sufficient number of sequential neighbors, the skewed overlay structure is more robust against attacks than the balanced structure. This is because the balanced overlay is more likely to be partitioned when a large number of nodes fails. Our results show that the combination of PNS and sufficient sequential neighbors can strike a balance between attack resilience and lookup performance.

7. Impact on Lookup Performance under Churn

In the original set of DHT proposals, lookups use *greedy routing*: routing from the source to the destination takes $O(\log N)$ hops which are in exponentially decreasing order of their spans (i.e., the length of the identifier space a hop spans). During routing, the choice of next hop at each step is deterministic, i.e., to pick the available neighbor which is the closest to the destination in the identifier space. Recent work [3] proposes *non-greedy routing* by which lookups have some freedom in route selection. Given a set of neighbors and a destination, the choice of next hop is flexible and can be made according to some metrics. Thus, we propose a new route selection algorithm, called biased route selection (BRS).

7.1. BRS

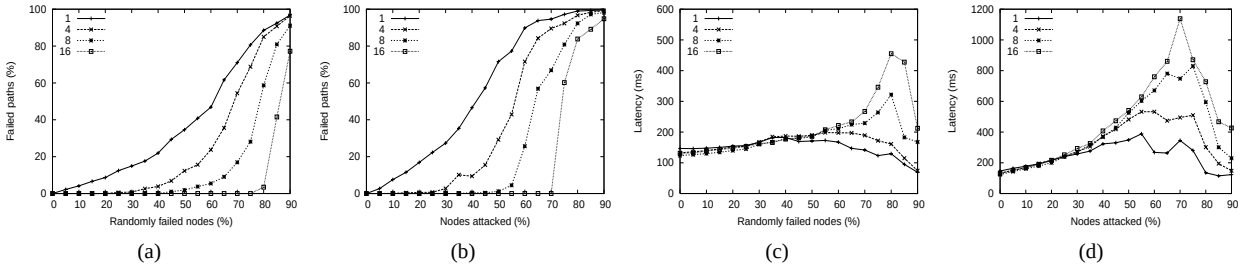
To make the choice of next hop, BRS bases its decision on both the neighbor's liveness probability estimates and latencies. Suppose a Chord node s wants to send a packet to the destination node t. Let n_i $(i = 1, 2, \cdots, k)$ be s's neighbors preceding t on the Chord ring, where $n_1 > n_2 > \cdots > n_k$ (node n_{j+1} precedes node n_j on the Chord ring). By greedy routing, s will deterministically send the packet to n_1 [7]. By BRS, in contrast, s prefers to send the packet to the neighbor n_i that has the largest value for the following equation

$$b_i = \frac{q_i}{latency(s, n_i)} \tag{4}$$

where q_i is the node n_i's liveness probability indicator.

As stated in Equation 4, BRS accounts for both neighbor's liveness and proximity. The intuition behind BRS is two-fold: (1) By forwarding packets to the neighbor which is most likely alive, BRS attempts to minimize timeouts due to neighbor failures. (2) By forwarding packets to the nearby neighbor, timeouts are quick upon node failures and lookup latency may be improved through low-latency links without the cost of increased hops.

[7] If n_1 fails, then s will try the alternate neighbor n_2.



(a) (b) (c) (d)

Figure 5. Results for varying number of sequential neighbors in PNS. (a) Percentage of failed paths for varying percentage of randomly failed nodes. (b) Percentage of failed paths for varying percentage of attacked nodes. (b) Average successful lookup latency for varying percentage of randomly failed nodes. (c) Average successful lookup latency for varying percentage of attacked nodes.

7.2. Simulation Methodology

We fix the number of sequential neighbors at 16 in p2psim. The basic stabilization interval is fixed at 18 seconds, while both the finger stabilization interval and successor list stabilization interval vary from 18 seconds to 19 minutes. In the simulated network, the number of nodes is 1024. Each node alternately leaves and rejoins the network. The interval between successive events for each node follows a Pareto distribution with median time of 1 hour (i.e., $\alpha = 1$ and $\beta = 1800$ sec). Each node issues lookups for random keys at intervals exponentially distributed with 116 seconds. Each simulation runs for 24 hours of simulated time. Statistics are collected only during the last 18 hours of the simulation. To account for bandwidth cost in p2psim, the size of a message is counted as 20 bytes (for packet overhead) plus 4 bytes for each IP address or node identifier or random key contained in the message.

In our simulations, lookups are performed using the *recursive* mode. A lookup is considered a failure if it returns the wrong node among the current set of participating nodes at the time the sender receives the lookup response, or if the sender receives no response within some timeout window [6]. To deal with lookup timeouts, p2psim timeouts individual messages after an interval of 3 times the round-trip time to the target node. p2psim retries lookups for up to a maximum of 4 seconds, after which p2psim declares the lookup has failed. Following the approach in [6], we use three metrics to characterize lookup performance: (1) *Median successful lookup latency*: the median latency of successful lookups. (2) *Lookup success rate*: the portion of successful lookups. (3) *Bandwidth cost (bytes/node/s)*: the bandwidth cost for all messages sent by a node, including periodic routing table refreshing traffic, lookup traffic, and join traffic. It is defined as the number of bytes per node per alive second.

7.3. Experimental Results

Performance comparison. Figure 6 shows the median successful lookup latency of various neighbor selection policies for different stabilization intervals. Several observations can be drawn from this figure: (1) Among all these neighbor selections, LNS achieves the lowest median lookup latency. This is mainly because LNS picks the node as neighbor which is likely to stay alive longer. The stable neighbor links result in less timeouts (and thus less retries on alternate neighbors) due to less dead routing table entries in LNS than other policies including PNS, DNS and RNS. This also suggests that timeouts are a significant component in lookup latencies under churn. (2) When the routing table refreshing interval is short ($\leq$72 sec), both PNS and LNS achieve lower lookup latencies than DNS and RNS. This is because the recovery process will quickly detect node failures and recover the routing table entries. Considering proximity in neighbor choice improves lookup performance. However, the short refresh interval results in high bandwidth consumption devoted to the maintenance messages as shown in Table 1. In addition, the recovery process will compete with user lookups for bandwidth. This accounts for lookup performance improvements by LNS over

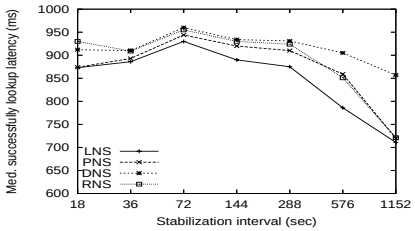


Figure 6. Median successful lookup latency of various neighbor selections for different stabilization intervals. Greedy routing is used in lookups.

Table 1. Successful lookup rate and bandwidth cost for LNS and PNS.

Stab. interval (sec)	LNS (bw_cost : rate)	PNS (bw_cost : rate)
18	14.2 : 0.998	14.2 : 0.998
36	11.7 : 0.998	11.8 : 0.998
72	10.5 : 0.998	10.6 : 0.998
144	9.7 : 0.998	9.9 : 0.998
288	9.3 : 0.998	9.3 : 0.998
576	8.7 : 0.998	8.8 : 0.998
1152	8.1 : 0.997	8.3 : 0.997

the other three policies. In LNS, less messages will be issued by the recovery process to find the replacements for the obsolete routing table entries. (3) When the refreshing interval is long (from 72 to 576 sec), the lookup latency improvement achieved by LNS is significant. This is because LNS picks more stable nodes as neighbors and thus incurs less timeouts for lookups, while obsolete routing entries in other policies cannot be discovered and repaired timely by the recovery process. (4) When the refreshing interval is sufficiently long (i.e., 1152 sec), many routing table entries would be obsolete due to churn without timely recovery. Surprisingly, the lookup latency instead decreases significantly. We give the following explanations. As shown in Table 1 (we omit the results for DNS and RNS due to space limitation), the refreshing interval of 1152 sec reduces the lookup success rate from 0.998 to 0.997, filtering the high-latency lookups that are considered failed lookups by timeouts.

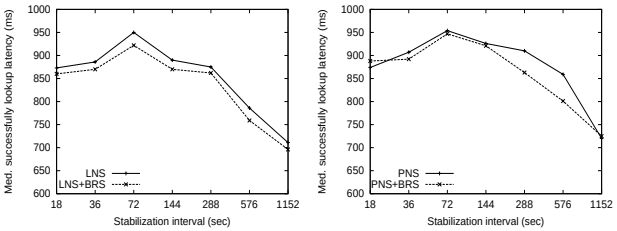


Figure 7. Impact of BRS on median successful lookup latency for LNS and PNS.

Effect of BRS. Figure 7 shows the impact of BRS on LNS and PNS with respect to stabilization intervals (we omit the results for DNS and RNS due to space constraints). Note that BRS improves lookup latency by up to 50ms. Also, our results show that BRS has little impact on bandwidth cost and lookup success rate for all the neighbor selections.

To summarize, LNS allows DHTs to choose a relatively long refreshing interval and thus consumes less bandwidth while achieving low lookup latencies. Long refreshing intervals mitigate the intense competition for bandwidth between the DHT maintenance and user lookups. BRS helps improve lookup latency under churn for all neighbor selection policies. Moreover, the snapshots of node degree distribution for LNS during simulations (omitted here due to space constraints), show that the in-degree distribution across nodes is very skewed. This is because the neighbor choice is based on the node liveness probability estimate — that is, nodes that have been alive for a long time are more likely to be others' neighbors. The skewed overlay structure produced by LNS makes us to believe that LNS should share similar characteristics in static resilience with PNS and DNS.

8. Conclusions

The issues we have discussed in this work fall into three categories: impact of neighbor selection on DHT overlay structure, on static resilience to random node failures and targeted node attacks, and on lookup performance under churn. We have presented LNS and BRS to improve lookup performance under churn. Our basic findings are: (1) Neighbor selections such as PNS may create an unbalanced overlay structure in terms of node in-degree distribution. (2) The unbalanced overlay structure resulting from neighbor selection may weaken the static resilience of DHTs to targeted node attacks. Surprisingly, the addition of sufficient sequential neighbors makes the unbalanced overlay more robust to targeted node attacks than the relatively balanced overlay created by RNS. (3) PNS exhibits less performance gain at the churn rate as those observed in deployed P2P systems. We confirm that timeouts are very critical to lookup latency in the face of churn. LNS achieves much better lookup performance under churn than current neighbor selection policies including PNS. Moreover, BRS can improve lookup latency by up to 50ms under churn for all neighbor selection policies. We hope the discussions in this work will provide some pieces of insights in DHT routing-level designs.

References

[1] B.-G. Chun, B. Y. Zhao, and J. D. Kubiatowicz. Impact of neighbor selection on performance and resilience of structured P2P networks. In *Proceedings of IPTPS*, Feb. 2005.

[2] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceeding of USENIX NSDI*, Mar. 2004.

[3] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, Aug. 2003.

[4] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of SIGCOMM Internet Measurement Workshop*, Nov. 2002.

[5] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of USENIX NSDI*, May 2005.

[6] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of IEEE INFOCOM*, March 2005.

[7] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of USENIX Technical Conference*, June 2004.

[8] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware*, Nov. 2001.

[9] S. Saroiu, K. P. Gummadi, and S. D. Gribble. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems Journal*, 9(2):170–184, Aug. 2003.

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