Proceedings of the General Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

Handling Churn in a DHT

Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz
University of California, Berkeley and Intel Research, Berkeley
{*srhea,geels,kubitron*}*@cs.berkeley.edu*, *troscoe@intel-research.net*

Abstract

This paper addresses the problem of *churn*—the continuous process of node arrival and departure—in distributed hash tables (DHTs). We argue that DHTs should perform lookups quickly and consistently under churn rates at least as high as those observed in deployed P2P systems such as Kazaa. We then show through experiments on an emulated network that current DHT implementations cannot handle such churn rates. Next, we identify and explore three factors affecting DHT performance under churn: reactive versus periodic failure recovery, message timeout calculation, and proximity neighbor selection. We work in the context of a mature DHT implementation called *Bamboo*, using the ModelNet network emulator, which models in-network queuing, cross-traffic, and packet loss. These factors are typically missing in earlier simulation-based DHT studies, and we show that careful attention to them in Bamboo's design allows it to function effectively at churn rates at or higher than that observed in P2P file-sharing applications, while using lower maintenance bandwidth than other DHT implementations.

1 Introduction

The popularity of widely-deployed file-sharing services has recently motivated considerable research into peer-to-peer systems. Along one line, this research has focused on the design of better peer-to-peer algorithms, especially in the area of structured peer-to-peer overlay networks or distributed hash tables (e.g. [20, 22, 24, 27, 30]), which we will simply call DHTs. These systems map a large identifier space onto the set of nodes in the system in a deterministic and distributed fashion, a function we alternately call *routing* or *lookup*. DHTs generally perform these lookups using only $O(\log N)$ overlay hops in a network of N nodes where every node maintains only $O(\log N)$ neighbor links, although recent research has explored the tradeoffs in storing more or less state.

A second line of research into P2P systems has focused on observing deployed networks (e.g. [5, 9, 13, 25]). A significant result of this research is that such networks are characterized by a high degree of churn. One metric of churn is node *session time*: the time between when a node joins the network until the next time it leaves. Median session times observed in deployed networks range from as long as an hour to as short as a few minutes.

In this paper we explore the performance of DHTs in such dynamic environments. DHTs may be better able to locate rare files than existing unstructured peer-to-peer networks [18]. Moreover, it is not hard to imagine that other proposed uses for DHTs will show similar churn rates to file-sharing networks—application-level multicast of a low-budget radio stream, for example. In spite of this promise, we show that short session times cause a variety of negative effects on two mature DHT implementations we tested. Both systems exhibit dramatic latency growth when subjected to increasing churn, and in one implementation the network eventually partitions, causing subsequent lookups to return inconsistent results. The remainder of this paper is dedicated to determining whether a DHT can be built such that it continues to perform well as churn rates increase.

We demonstrate that DHTs can in fact handle high churn rates, and we identify and explore several factors that affect the behavior of DHTs under churn. The three most important factors we identify are:

- reactive versus periodic recovery from failures
- calculation of message timeouts during lookups
- choice of nearby over distant neighbors

By *reactive recovery*, we mean the strategy whereby a DHT node tries to find a replacement neighbor immediately upon noticing that an existing neighbor has failed. We show that under bandwidth-limited conditions, reactive recovery can lead to a positive feedback cycle that overloads the network, causing lookups to have high latency or to return inconsistent results. In contrast, a DHT node may recover from neighbor failure at a fixed, periodic rate. We show that this strategy improves performance under churn by allowing the system to avoid positive feedback cycles.

The manner in which a DHT chooses timeout values during lookups can also greatly affect its performance under churn. If a node performing a lookup sends a message to a node that has left the network, it must eventually timeout the request and try another neighbor. We demonstrate that such timeouts are a significant component of lookup latency under churn, and we explore several methods of computing good timeout values, including virtual coordinate schemes as used in the Chord DHT.

Finally, we consider *proximity neighbor selection* (PNS), where a DHT node with a choice of neighbors tries to select those that are most nearby itself in network latency. We compare several algorithms for discovering nearby neighbors—including algorithms similar to those used in the Chord, Pastry, and Tapestry DHTs—to show the tradeoffs they offer between latency reduction and added bandwidth.

We have augmented the Bamboo DHT [23] such that it can be configured to use any of the design choices described above. As such, we can examine each design decision independently of the others. Moreover, we examine the performance of each configuration by running it on a large cluster with an emulated wide-area network. This methodology is particularly important with regard to the choice of reactive versus periodic recovery as described above. Existing studies of churn in DHTs (e.g. [7, 8, 16, 19]) have used simulations that—unlike our emulated network—did not model the effects of network queuing, cross traffic, or message loss. In our experience, these effects are primary factors contributing to DHTs' inability to handle churn. Moreover, our measurements are conducted on an isolated network, where the only sources of queuing, cross traffic, and loss are the DHTs themselves; in the presence of heavy background traffic, we expect that such network realities will exacerbate the ability of DHTs to handle even lower levels of churn.

Of course, this study has limitations. Building and testing a complete DHT implementation on an emulated network is a major effort. Consequently, we have limited ourselves to studying a single DHT on a single network topology using a relatively simple churn model. Furthermore, we have not yet studied the effects of some implementation decisions that might affect the performance of a DHT under churn, including the use of alternate routing table neighbors as in Kademlia and Tapestry, or the use of iterative versus recursive routing. Nevertheless, we believe that the effects of the factors we have studied are dramatic enough to present them as an important early study in the effort to build a DHT that successfully handles churn.

The rest of this paper is structured as follows: in the next section we review how DHTs perform routing or lookup, with particular reference to Pastry, whose routing algorithm Bamboo also uses. In Section 3, we review existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT implementations under such churn. In Section 4, we study each of the factors listed above in isolation, and describe how Bamboo uses these techniques. In Section 5, we survey related work, and in Section 6 we discuss important future work. We conclude in Section 7.

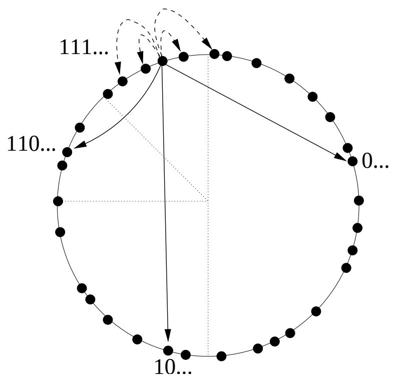


Figure 1: *Neighbors in Pastry and Bamboo.* A node's neighbors are divided into its *leaf set*, shown as dashed arrows, and its *routing table*, shown as solid arrows.

2 Introduction to DHT Routing

In this section we present a brief review of DHT routing, using Pastry [24] as an example. The *geometry* and *routing algorithm* of Bamboo are identical to Pastry; the difference (and the main contribution of this paper) lies in how Bamboo maintains the geometry as nodes join and leave the network and the network conditions vary.

DHTs are structured graphs, and we use the term geometry to mean the pattern of neighbor links in the overlay network, independent of the routing algorithms or state management algorithms used [12].

Each node in Pastry is assigned a numeric identifier in $[0, 2^{160})$, derived either from the SHA-1 hash of the IP address and port on which the node receives packets or from the SHA-1 hash of a public key. As such, they are well-distributed throughout the identifier space.

In Pastry, a node maintains two sets of neighbors, the *leaf set* and the *routing table* (see Figure 1). A node's leaf set is the set of $2k$ nodes immediately preceding and following it in the circular identifier space. We denote this set by L, and we use the notation L_i with $-k \leq i \leq k$ to denote the members of L, where L_0 is the node itself.

In contrast, the routing table is a set of nodes whose identifiers share successively longer prefixes with the source node's identifier. Treating each identifier as a sequence of digits of base 2^b and denoting the routing table entry at row l and column i by $R_l[i]$, a node chooses its neighbors such that the entry at $R_l[i]$ is a node whose identifier matches its own in exactly l digits and whose

```
if ($L_{-k} \leq D \leq L_k$)
    next_hop = $L_i$ s.t. $|D - L_i|$ is minimal
else if ($R_l[D[l]] \neq \mathbf{null}$)
    next_hop = $R_l[D[l]]$
else
    next_hop = $L_i$ s.t. $|D - L_i|$ is minimal
```

Figure 2: *The Bamboo routing algorithm.* The code shown chooses the next routing hop in for a message with destination D, where D matches the identifier of the local node in the first l digits.

$(l + 1)$th digit is i. In the experiments in this paper, Bamboo uses binary digits ($b = 1$), though it can be configured to use any base.

The basic operation of a DHT is to consistently map identifiers onto nodes from any point in the system, a function we call *routing* or *lookup*. Pastry achieves consistent lookups by directing each identifier to the node with the numerically closest identifier. Algorithmically, routing proceeds as shown in Figure 2. To route a message with key D, a node first checks whether D lies within its leaf set, and if so, forwards it to the numerically closest member of that set (modulo 2^{160}). If that member is the local node, routing terminates. If D does not fall within the leaf set, the node computes the length l of the longest matching prefix between D and its own identifier. Let $D[i]$ denote the ith digit of D. If $R_l[D[l]]$ is not empty, the message is forwarded on to that node. If neither of these conditions is true, the message is forwarded to the member of the node's leaf set numerically closest to D. Once the destination node is reached, it sends a message back to the originating node with its identifier and network address, and the lookup is complete.

We note that a node can often choose between many different neighbors for a given entry in its routing table. For example, a node whose identifier begins with a 1 needs a neighbor whose identifier begins with a 0, but such nodes make up roughly half of the total network. In such situations, a node can choose between the possible candidates based on some metric. *Proximity neighbor selection* is the term used to indicate that nodes in a DHT use network latency as the metric by which to choose between neighbor candidates.

Using this design, Pastry and Bamboo perform lookups in $O(\log N)$ hops [24], while the leaf set allows forward progress (in exchange for potentially longer paths) in the case that the routing table is incomplete. Moreover, the leaf set adds a great deal of *static resilience* to the geometry; Gummadi et al. [12] show that with a leaf set of 16 nodes, even after a random 30% of the links are broken there are still connected paths between all node pairs in a network of 65,536 nodes. This resilience is important in handling failures in general and churn in particular, and was the reason we chose the Pastry geometry for use in Bamboo. We could also have used a pure ring geometry as in Chord, extending it to account for proximity in neighbor selection as described in [12].

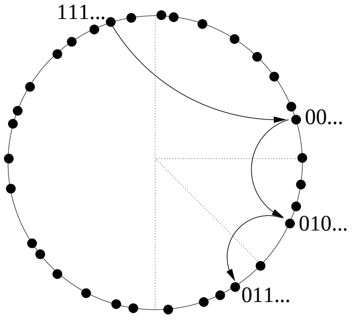


Figure 3: *Recursive lookup.* To find the node closest to identifier 011, the node whose identifier starts with 111 sends a lookup message to its neighbor whose first digit is 0. This node then forwards the query to its neighbor whose first two digits are 01, and from there the node is forwarded to the neighbor whose first three digits are 011.

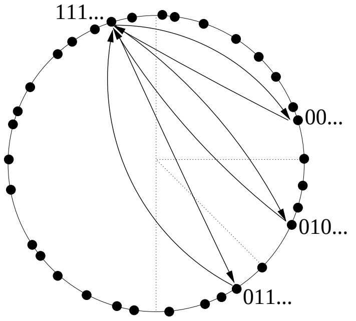


Figure 4: *Iterative lookup.* An iterative lookup involves the same nodes as a recursive one, but instead of forwarding the message, each intermediate node responds to the source with the address of the next hop.

The manner in which we have described routing so far is commonly called *recursive* routing (Figure 3). In contrast, lookups may also be performed *iteratively*. As shown in Figure 4, an iterative lookup involves the same nodes as a recursive one, but the entire process is controlled by the source of the lookup. Rather than asking a neighbor to forward the lookup through the network on its behalf, the source asks that neighbor for the network address of the next hop. The source then asks the newly-discovered node the same question, repeating the process until no further progress can be made, at which point the lookup is complete.

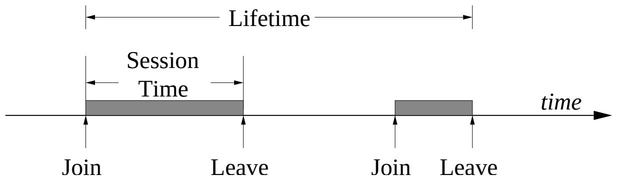


Figure 5: *Metrics of churn.* With respect to the routing and lookup functionality of a DHT, the *session times* of nodes are more relevant than their *lifetimes*.

3 The Problem of Churn

There have been very few large-scale DHT-based application deployments to date, and so it is hard to derive good requirements on churn-resilience. However, P2P file-sharing networks provide a useful starting point. These systems provide a simple indexing service for locating files on those peer nodes currently connected to the network, a function which can be naturally mapped onto a DHT-based mechanism. For example, the Overnet file-sharing system uses the Kademlia DHT to store such an index. While some DHT applications (such as file storage as in CFS [10]) might require greater client availability, others may show similar churn rates to file-sharing networks (such as end-system multicast or a rendezvous service for instant messaging). As such, we believe that DHTs should at least handle churn rates observed in today's file-sharing networks. To that end, in this section we survey existing studies of churn in deployed file-sharing networks, describe the way we model such churn in our emulated network, and quantify the performance of mature DHT implementations under such churn.

Studies of existing file-sharing systems mainly use two metrics of churn (see Figure 5). A node's *session time* is the elapsed time between it joining the network and subsequently leaving it. In contrast, a node's *lifetime* is the time between it entering the network for the first time and leaving the network permanently. The sum of a node's session times divided by its lifetime is often called its *availability*. One representative study [5] observed median session times on the order of tens of minutes, median lifetimes on the order of days, and median availability of around 30%.

With respect to the lookup functionality of a DHT, we argue that session time is the most important metric. Even temporary loss of a routing neighbor weakens the correctness and performance guarantees of a DHT, and unavailable neighbors reduce a node's effective connectivity, forcing it to choose suboptimal routes and increasing the destructive potential of future failures. Since nodes are often unavailable for long periods, remembering neighbors that have failed is of little value in performing lookups. While remembering neighbors is useful for applications like storage [6], it is of little value for *lookup* operations.

First Author	Systems Observed	Session Time
Saroiu [25]	Gnutella, Napster	50% $\leq$ 60 min.
Chu [9]	Gnutella, Napster	31% $\leq$ 10 min.
Sen [26]	FastTrack	50% $\leq$ 1 min.
Bhagwan [5]	Overnet	50% $\leq$ 60 min.
Gummadi [13]	Kazaa	50% $\leq$ 2.4 min.

Table 1: *Observed session times in various peer-to-peer systems.* The median session time ranges from an hour to a minute.

3.1 Empirical studies

Elsewhere [23] we have surveyed published studies of deployed file-sharing networks. Table 1 shows a summary of observed session times. At first sight, some of these values are surprising, and may be due to methodological problems with the study in question or malfunctioning of the system under observation. However, it is easy to image a user joining the network, downloading a single file (or failing to find it), and leaving, making session times of a few minutes at least plausible. To be conservative, then, we would like a DHT to be robust for median session times from as much as an hour to as little as a minute.

3.2 Experimental Methodology

Our platform for measuring DHT performance under churn is a cluster of 40 IBM xSeries PCs, each with Dual 1GHz Pentium III processors and 1.5GB RAM, connected by Gigabit Ethernet, and running either Debian GNU/Linux or FreeBSD. We use ModelNet [28] to impose wide-area delay and bandwidth restrictions, and the Inet topology generator [3] to create a 10,000-node wide-area AS-level network with 500 client nodes connected to 250 distinct stubs by 1 Mbps links. To increase the scale of the experiments without overburdening the capacity of ModelNet by running more client nodes, each client node runs two DHT instances, for a total of 1,000 DHT nodes.

Our control software uses a set of wrappers which communicate locally with each DHT instance to send requests and record responses. Running 1000 DHT instances on this cluster (12.5 nodes/CPU) produces CPU loads below one, except during the highest churn rates. Ideally, we would measure larger networks, but 1000-node systems already demonstrate problems that will surely affect larger ones.

In an experiment, we first bring up a network of 1000 nodes, one every 1.5 seconds, each with a randomly assigned gateway node to distribute the load of bootstrapping newcomers. We then churn nodes until the system performance levels out; this phase normally lasts 20-30 minutes but can take an hour or more. Node deaths are timed by a Poisson process and are therefore uncorrelated and bursty. A new node is started each time one is killed, maintaining the total network size at 1000. This model of churn is similar to that described by Liben-Nowell et all [17]. In a Poisson process, an event rate λ corresponds to a median inter-event period of $\ln 2/\lambda$. For each event we select a node to die uniformly at random, so each node's session time is expected to span N events, where N is the network size. Therefore a churn rate of λ corresponds to a median node session time of

$$t_{\text{med}} = N \ln 2/\lambda.$$

For example, a 1000-node network churning with median session times of one hour will see one node arrive (and one leave) every 5.2 seconds. In our experiments, we used churn rates ranging from 8/second to 4/minute, equal to median session times from 1.4 minutes to 3 hours.

Each live node continually performs lookups for identifiers chosen uniformly at random, timed by a Poisson process with rate 0.1/second, for an aggregate system load of 100 lookups/second. Each lookup is simultaneously performed by ten nodes, and we report both whether it completes and whether it is consistent with the others for the same key. If there is a majority among the ten results for a given key, all nodes in the majority are said to see a consistent result, and all others are considered inconsistent. If there is no majority, all nodes are said to see inconsistent results. This metric of consistency is more strict than that required by some DHT applications. However, both MIT's Chord and our Bamboo implementation show at least 99.9% consistency under 47-minute median session times [23], so it does not seem unreasonable.

There are two ways in which lookups fail in our tests. First, we do not perform end-to-end retries, so a lookup may fail to complete if a node in the middle of the lookup path leaves the network before forwarding the lookup request to the next node. We observed this behavior primarily in FreePastry as described below. Second, a lookup may return inconsistent results. Such failures occur either because a node is not aware of the correct node to forward the lookup to, or because it erroneously believes the correct node has left the network (because of congestion or poorly chosen timeouts). All DHT implementations we have tested show some inconsistencies under churn, but carefully chosen timeouts and judicious bandwidth usage can minimize them.

3.3 Existing DHTs

In this section we report the results of testing two mature DHT implementations under churn. Our intent here is not to place a definitive bound on the performance of either implementation. Rather, it is to motivate our work by demonstrating that handling churn in DHTs is both an important and a non-trivial problem. While we have discussed these experiments extensively with the authors of both systems, it is still possible that alternative configurations could have improved their performance. Moreover, both systems have seen subsequent development, and newer versions may show improved resilience under churn.

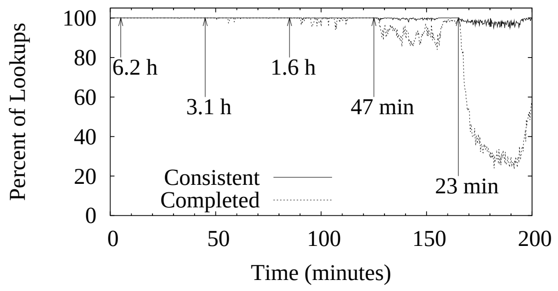


Figure 6: *FreePastry under churn.* The percentage of successful lookups in a 1000-node FreePastry network under churn. Session times for each 30-minute churn period are indicated by arrows, and each churn period is separated from the next by 10 minutes of no churn. The churn rate doubles with each successive period.

FreePastry We tested FreePastry 1.3, the Rice University implementation of Pastry [1]. Figure 6 shows one effect of churn on a network of 1000 FreePastry nodes, which we ran using the default 24-node leaf sets and logarithm base of 16. We do not enforce proximity between a new node and its gateway, as suggested for best FreePastry performance; this decision only effects the proximity of a node's neighbors, not the efficiency of its routing.

It is clear from Figure 6 that while successful lookups are mostly consistent, FreePastry fails to complete a majority of lookup requests under heavy churn. A likely explanation for this failure is that nodes wait so long on lookup requests to time out that they frequently leave the network with several requests still in their queues. This behavior is probably exacerbated by FreePastry's use of Java RMI over TCP as its message transport, and the way that FreePastry nodes handle the loss of their neighbors. We present evidence to support these ideas in Section 4.1.

We make a final comment on this graph. FreePastry generally recovers well between churn periods, once again correctly completing all lookups. The difficulty with real systems is that there is no such quiet period; the network is in a continual state of churn.

MIT Chord We tested MIT's Chord implementation [4] using a CVS snapshot from 8/4/2003, with the default 10-node successor lists and with the location cache disabled (using the `-F` option), since the cache causes poor performance under churn.

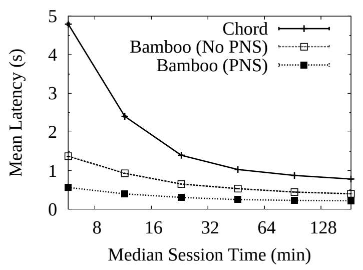


Figure 7: *Chord under churn.* Shown is the mean latency of lookups in a 1000-node MIT Chord network under increasing levels of churn. Churn increases to the left.

In contrast to FreePastry, almost all lookups in a Chord network complete and return consistent results. Chord's shortcoming under churn is in lookup latency, as shown in Figure 7, which shows the result of running Chord under the same workload as shown in Figure 6, but where we have averaged the lookup latency over each churn period. Shown for comparison are two lines representing Bamboo's performance in the same test, with and without proximity neighbor selection (PNS). Under all churn rates, Bamboo is using slightly under 750 bytes per second per node, while Chord is using slightly under 2,400.

We discuss in detail the differences that enable Bamboo to outperform Chord in Sections 4.2 and 4.3, but some of the difference in latency between Bamboo and Chord is due to their routing styles. Bamboo performs lookups recursively, whereas Chord routes iteratively. Chord could easily be changed to route recursively; in fact, newer versions of Chord support both recursive routing and PNS. Note, however, that Chord's latency grows more quickly under increasing churn than does Bamboo's. In Section 4.2, we will show evidence to support our belief that this growth is due to Chord's method of choosing timeouts for lookup messages and is independent of the lookup style employed.

3.3.1 Summary

To summarize this section, we note that we have observed several effects of churn on existing DHT implementations. A DHT may fail to complete lookup requests altogether, or it may complete them but return inconsistent results for the same lookup launched from different source nodes. On the other hand, a DHT may continue to return consistent results as churn rates increase, but it may suffer from a dramatic increase in lookup latency in the process.

4 Handling Churn

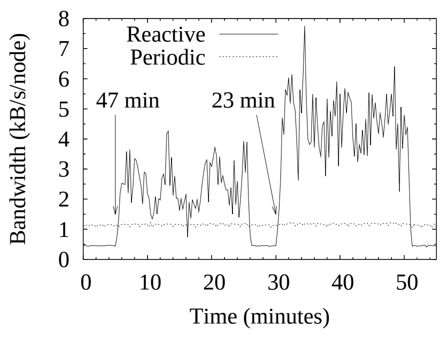
Having briefly described the way in which DHTs perform lookups, and having given evidence indicating that their ability to do so is hindered under churn, we now turn to the heart of this paper: a study of the factors contributing to this difficulty, and a comparison of solutions that can be used to overcome them. In turn, we discuss reactive versus periodic recovery from neighbor failure, the calculation of good timeout values for lookup messages, and techniques to achieve proximity in neighbor selection. The remainder of this paper focuses only on the Bamboo DHT, in which we have implemented each alternative design choice studied here. Working entirely within a single implementation allows us to minimize the differences between experiments comparing one design choice to another.

4.1 Reactive vs. Periodic Recovery

Early implementations of Bamboo suffered performance degradation under churn similar to that of FreePastry. MIT Chord's performance, however, does not degrade in the same way. A significant difference in its behavior is a design choice about how to handle detected node failures. We will call the two alternative approaches reactive and periodic recovery.

Reactive recovery In reactive recovery, a node reacts to the loss of one if its existing leaf set neighbors (or the appearance of a new node that should be added to its leaf set) by sending a copy of its new leaf set to every node in it. To save bandwidth, a node can only send differences from the last message, but the total number of messages is still $O(k^2)$ for a leaf set of k nodes. This algorithm converges quickly, is used in FreePastry, and was used in early versions of Bamboo. MSPastry uses a more bandwidth-efficient, but more complex, variant of reactive recovery [7].

Periodic recovery In contrast, in periodic recovery a node periodically shares its leaf set with each of the members of that set, each of whom responds in kind with its own leaf set. The process takes place independently of the node detecting changes in its leaf set. As a simple optimization, a node picks one random member of its leaf set to share state with in each period. This change saves bandwidth, but still converges in $O(\log k)$ phases, where k is the size of the leaf set. (Further details can be found elsewhere [23].) This algorithm is the one currently used by Bamboo, and the periodic nature of this algorithm is shared by Chord's method of keeping its successor list correct.



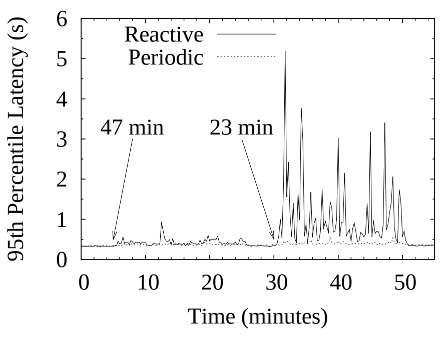


Figure 8: *Reactive versus periodic recovery.* Without churn, reactive recovery is very efficient, as messages are only sent in response to actual changes. At reasonable churn rates, however, periodic recovery uses less bandwidth, and lower contention for the network leads to lower latencies.

4.1.1 Positive feedback cycles

Reactive recovery runs the risk of creating a positive feedback cycle as follows. Consider a node whose access link to the network is sufficiently congested that timeouts cause it to believe that one of its neighbors has failed. If the node is recovering reactively, recovery operations begin, and the node will add even more packets to its already congested network link. This added congestion will increase the likelihood that the node will mistakenly conclude that other neighbors have failed. If this process continues, the node will eventually cause congestion collapse on its access link.

Observations of these cycles in early Bamboo (and examination of the Chord code) originally led us to propose periodic recovery for handling churn. By decoupling the rate of recovery from the discovery of failures, periodic recovery prevents the feedback cycle described above. Moreover, by lengthening the recovery period with the observation of message timeouts, we can introduce a negative feedback cycle, further improving resilience.

Another way to mitigate the instability associated with reactive recovery is to be more conservative when detecting node failure. We have found one effective approach to be to conclude failure only after 15 consecutive message timeouts to a neighbor. Since timeouts are backed off multiplicatively to a maximum of five seconds, it is unlikely that a node will conclude failure due to congestion. One drawback with this technique, however, is that neighbors that have actually failed remain in a node's routing table for some time. Lookups that would route through these neighbors are thus delayed, resulting in long lookup latencies. To remedy this problem, a node stops routing through a neighbor after seeing five consecutive message timeouts to that neighbor. We have found these changes make reactive recovery feasible for small leaf sets and moderate churn.

4.1.2 Scalability

Experiments show little difference in correctness between periodic and reactive recovery. To see why, consider a node A that joins a network, and let B be the node in the existing network whose identifier most closely matches that of A. As in Pastry, A retrieves its initial leaf set by contacting B, and B adds A to its leaf set immediately after confirming its IP address and port (with a probe message). Until A's arrival propagates through the network, another node C may still route messages that should go to A to B instead, but B will just forward these messages on to A. Likewise, should A fail, B will still be in C's leaf set, so once routing messages to A time out, C and other nearby nodes will generally all agree that B is the next best choice.

While both periodic and reactive recovery achieve roughly identical correctness, there is a large difference in the bandwidth consumed under different churn rates and leaf set sizes. (A commonly accepted rule of thumb is that to provide sufficient resilience to massive node failure, the size of a node's leaf set should be logarithmic in the system size.) Under low churn, reactive recovery is very efficient, as messages are only sent in response to actual changes, whereas periodic recovery is wasteful. As churn increases, however, reactive recovery becomes more expensive, and this behavior is exacerbated by increasing leaf set size. Not only does a node see more failures when its leaf set is larger, but the set of other nodes it must notify about the resulting changes in its own leaf set is larger. In contrast, periodic recovery aggregates all changes in each period into a single message.

Figure 8 shows this contrast in Bamboo using leaf sets of 24 nodes, the default leaf set size in FreePastry. In this figure, we ran Bamboo using both configurations for two 20-minute churn periods of 47 and 23 minute median session times separated by five minutes with no churn.

We note that during the periods of the test where there is no churn, reactive recovery uses less than half of the bandwidth of periodic recovery. On the other hand, under churn its bandwidth use jumps dramatically. As discussed above, Bamboo does not suffer from positive feedback cycles on account of this increased bandwidth usage. Nevertheless, the extra messages sent by reactive recovery compete with lookup messages for the available bandwidth, and as churn increases we see a corresponding increase in lookup latency. Although not shown in the figure, the number of hops per lookup is virtually identical between the two schemes, implying that the growth in bandwidth is most likely due to contention for the available bandwidth.

Since our goal is to handle median session times down to a few minutes with low lookup latency, we do not explore reactive recovery further in this work. The remainder of the Bamboo results we present are all obtained using periodic recovery.

4.2 Timeout Calculation

In this section, we discuss the role that timeout calculation on lookup messages plays in handling churn.

To understand the relative importance of timeouts in a DHT as opposed to a more traditional networked system, consider a traditional client-server system such as the networked file system (NFS). In NFS, the server does not often fail, and when it does there are generally few options for recovery and no alternative servers to fail over to. If a response to an NFS request is not received in the expected time, the client will usually try again with an exponentially increasing timeout value.

In a peer-to-peer system under churn, in contrast, requests will be frequently sent to a node that has left the system, possibly forever. At the same time, a DHT with routing flexibility (static resilience) has many alternate paths available to complete a lookup. Simply backing off the request period is thus a poor response to a request timeout; it is often better to retry the request through a different neighbor.

A node should ensure that the timeout for a request was judiciously selected before routing to an alternate neighbor. If it is too short, the node to which the original was sent may be yet to receive it, may be still processing it, or the response may be queued in the network. If so, injecting additional requests may result in the use of additional bandwidth without any beneficial result—for example, in the case that the local node's access link is congested. Conversely, if the timeout is too long, the requesting node may waste time waiting for a response from a node that has left the network. If the request rate is fixed at too low a value, these long waits cause unbounded queue growth on the request node that might be avoided with shorter timeouts.

For these reasons, nodes should accurately choose timeouts such that a late response is indicative of node failure, rather than network congestion or processor load.

4.2.1 Techniques

We discuss and study three alternative timeout calculation strategies. In the first, we fix all timeouts at a conservative value of five seconds as a control experiment. In the second, we calculate TCP-style timeouts using direct measurement of past response times. Finally, we explore using indirect measurements from a virtual coordinate algorithm to calculate timeouts.

TCP-style timeouts: If a DHT routes recursively, it rarely communicates with nodes other than its direct neighbors in the overlay network. Since the number of these neighbors is logarithmic in the size of the network, and since each node periodically probes each neighbor for availability, a node can easily maintain a past history of each neighbor's response times for use in calculating timeouts. In Bamboo, we have implemented this strategy following the style of the early TCP work [15], where each node maintains an exponentially weighted mean and variance of the response time for each neighbor. Specifically, the estimate round-trip timeout (RTO) for a neighbor is calculated as

$$\mathrm{RTO} = \mathrm{AVG} + 4 \times \mathrm{VAR},$$

where AVG is the observed average round-trip time and VAR is the observed mean variance of that time.

Timeouts from virtual coordinates: In contrast to recursive routing, with iterative routing a node must potentially have a good timeout for *any* other node in the network. However, in some scenarios iterative routing does have attractive properties. For example, since the source of a lookup request controls the entire process of iterative routing, it is easy to explore several different lookup paths in parallel. For only a constant increase in bandwidth used, this technique prevents a single timeout from delaying a lookup [16].

Virtual coordinates provide one approach to computing timeouts without previously measuring the response time to every node in the system. In this scheme, a distributed machine learning algorithm is employed to assign to each node coordinates in a virtual metric space such that the distance between two nodes in the space is proportional to their latency in the underlying network.

Bamboo includes an implementation of the Vivaldi coordinate system employed by Chord [11]. Vivaldi keeps an exponentially-weighted average of the error of past round-trip times calculated with the coordinates, and computes the RTO as

$$\mathrm{RTO} = v + 6 \times \alpha + 15$$

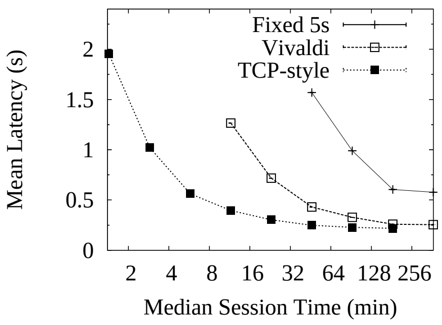


Figure 9: *TCP-style versus virtual coordinate-based timeouts in Bamboo.* Timeouts chosen using Vivaldi are competitive with TCP-style timeouts for moderate churn rates.

where v is the predicted round-trip time and α is the average error. The constant term of 15 milliseconds is added to avoid unnecessary retransmissions when the destination is the local host.

4.2.2 Results

TCP-style timeouts assume a recursive routing algorithm, and a virtual coordinate system is necessary only when routing iteratively. While we would ideally compare the two approaches by measuring each in its intended environment, this would prevent us from isolating the effect of timeouts from the differences caused by routing styles.

Instead, we study both schemes under recursive routing. If timeouts calculated with virtual coordinates provide performance comparable to those calculated in the TCP-style under recursive routing, we can expect the virtual coordinate scheme to not be prohibitively expensive under iterative routing. While other issues may remain with iterative routing under churn (e.g. congestion control—see Section 6), this result would be a useful one.

Figure 9 shows a direct comparison of the three timeout calculation methods under increasing levels of churn. In all cases in this experiment, the Bamboo configurations differed only in choice of timeout calculation method. Proximity neighbor selection was used, but the latency measurements for PNS used separate direct probing and not the virtual coordinates.

Even under light levels of churn, fixing all timeouts at five seconds causes lookup timeouts to pull the mean latency up to more than twice that of the other configurations, confirming our intuition about the importance of good timeout values in DHT routing under churn. Moreover, by comparing Figure 9 to Figure 7, we note that under high churn timeout calculation has a greater effect on lookup latency than the use of PNS.

Virtual coordinate-based timeouts achieve very similar mean latency to TCP-style timeouts at low churn. Furthermore, they perform within a factor of two of TCP-style measurements until the median churn rate drops to 23 minutes. Past this point, their performance quickly diverges, but virtual coordinates continue to provide mean lookup latencies under two seconds down to twelve-minute median session times.

Finally, we note the similarity in shape of Figure 9 to Figure 7, where we compared the performance of Chord to Bamboo, suggesting that the growth in lookup latency under Chord at high churn rates is due to timeout calculation based on virtual coordinates.

4.3 Proximity Neighbor Selection

Perhaps one of the most studied aspects of DHT design has been proximity neighbor selection (PNS), the process of choosing among the potential neighbors for any given routing table entry according to their network latency to the choosing node. This research is well motivated. The *stretch* of a lookup operation is defined as the latency of the lookup divided by the round-trip time between the lookup source and the node discovered by the lookup in the underlying IP network. Dabek et al. present an argument and experimental data that suggest that PNS allows a DHT of N nodes to achieve median stretch of only 1.5, independent of the size of the network and despite using $O(\log N)$ hops [11]. Others have proved that PNS can be used to provide constant stretch in locating replicas under a restricted network model [21]. This is the first study of which we are aware, however, to compare methods of achieving PNS under churn. We first take a moment to discuss the common philosophy and techniques shared by each of the algorithms we study.

4.3.1 Commonalities

One of the earliest insights in DHT design was the separation of correctness and performance in the distinction between neighbors in the leaf set and neighbors in the routing table [24, 27]. So long as the leaf sets in the system are correct, lookups will always return correct results, although they may take $O(N)$ hops to do so. Leaf set maintenance is thus given priority over routing table maintenance by most DHTs. In the same manner, we note that so long as each entry in the routing table has *some* appropriate neighbor (i.e. one with the correct identifier prefix), lookups will always complete in $O(\log N)$ hops, even though they make take longer than if the neighbors had been chosen for proximity. We say such lookups are *efficient*, even though they may not have low stretch. By this argument, we reason that it is desirable to fill a routing table entry quickly, even with a less than optimal neighbor; finding a nearby neighbor is a lower priority.

There is a further argument to treating proximity as a lower priority in the presence of churn. Since we expect our set of neighbors to change over time as part of the churn process, it makes little sense to work too hard to find the absolute closest neighbor at any given time; we might expend considerable bandwidth to find them only to see them leave the network shortly afterward. As such, our general approach is to run each of the algorithms described below *periodically*. In the case where churn is high, this technique allows us to retune the routing table as the network changes. When churn is low, rerunning the algorithms makes up for latency measurement errors caused by transient network conditions in previous runs.

Our general approach to finding nearby neighbors thus takes the following form. First, we use one of the algorithms below to find nodes that may be near to the local node. Next, we measure the latency to those nodes. If we have no existing neighbor in the routing table entry that the measured node would fill, or if it is closer than the existing routing table entry, we replace that entry, otherwise we leave the routing table unchanged. Although the bandwidth cost of multiple measurements is high, the storage cost to remember past measurements is low. As a compromise, we perform only a single latency measurement to each discovered node during any particular run of an algorithm, but we keep an exponentially weighted average of past measurements for each node, and we use this average value in deciding the relative closeness of nodes. This average occupies only eight bytes of memory for each measured node, so we expect this approach to scale comfortably to very large systems.

4.3.2 Techniques

The techniques for proximity neighbor selection that we study here are global sampling, sampling of our neighbors' neighbors, and sampling of the nodes that have our neighbors as their neighbors. We describe each of these techniques in turn.

Global sampling In global sampling (called *global tuning* in our earlier work [23]), we use the lookup functionality of the DHT to find new neighbors. For a routing table entry that requires a neighbor with prefix p, we perform a lookup for a random identifier with prefix p. The node returned by this lookup will almost always have the desired prefix. (As an example of why this is not always the case, note that a lookup of identifier 0 may return a node whose identifier starts with 1 if the node with the largest identifier in the ring is numerically closer to 0 than the node with the smallest identifier.) Given enough samples, all nodes with prefix p will eventually be probed. The motivation for this technique comes from Gummadi et al., who showed that sampling only around 16 nodes for each routing table entry provides almost optimal proximity [12].

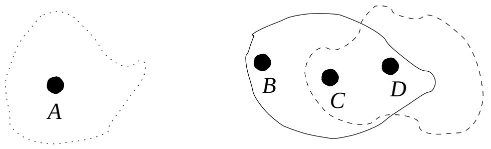


Figure 10: *Sampling neighbors' neighbors.* If A joins using D as its gateway, its initial level-0 neighbors are the same as those of D; assume that these are all within the dashed line. A contacts a level-0 neighbor, e.g. C, and asks it for its level-0 neighbors. A would learn about B in this manner. However, there may be no path from the D's ideal neighbors to those of A.

There are some cases, however, where global sampling will take unreasonably long to find the closest possible neighbor. For example, consider two nodes separated from the core Internet by the same, high latency access link, as shown in Figure 11. The relatively high latency seen by these two nodes to all other nodes in the network makes them attractive neighbors for each other; if they have different first digits in a network with logarithm base two, they can drastically reduce the cost of the first hop of many routes by learning about each other. However, the time for these nodes to find each other using global sampling is proportional to the size of the total network, and so they may not find each other before their sessions end. It is this drawback of global sampling that leads us to consider other techniques.

Neighbors' neighbors The next technique we consider is sampling our neighbors neighbors, a process called *routing table maintenance* in the Pastry work [24] or *local tuning* in our earlier work [23]. In this technique, we contact an existing routing table neighbor at level l of our routing table and ask for its level l neighbors. Like us, these nodes share a prefix of $l-1$ digits with the contacted neighbor and are thus appropriate for use in our routing table as well. As in global sampling, having discovered these new nodes, we probe them for latency and use them if they are closer than our existing neighbors.

The motivation for sampling neighbors' neighbors is illustrated in Figure 10; it relies on the expectation that proximity in the network is roughly transitive. If a node discovers one nearby node, then that node's neighbors are probably also nearby. In this way, we expect that a node can "walk" through the graph of neighbor links to the set of nodes most near it.

To see one possible shortcoming of sampling our neighbors' neighbors, consider again Figure 11. While the two isolated nodes would like to discover each other, it is unlikely that any other nodes in the network would prefer them as neighbors; their isolation makes them unattractive for routing lookups that originate elsewhere, except

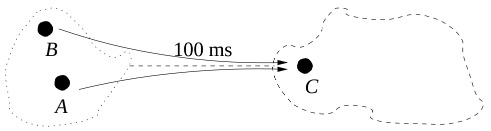


Figure 11: *Sampling neighbors' inverse neighbors.* Nodes A and B are isolated from the remainder of the network by a long latency, and are initially unaware of each other. Such a situation is possible if, for example, two European nodes join a network of primarily North American nodes. As such, they make unattractive neighbors for other nodes, but they would still like to find each other. If they both have C as a neighbor, they can find each other by asking C for its inverse neighbors.

in the rare case that they are the result of those lookups. As such, since neighbor links in DHTs are rarely symmetric, it is unlikely that there is a path through the graph of neighbor links that will lead one isolated node to the other, despite their relative proximity.

Neighbors' inverse neighbors The latter argument presents an obvious alternative approach. Instead of sampling our neighbors' neighbors, why not sample those nodes which have the same neighbors as the local node? This technique was originally proposed in the Tapestry nearest neighbor algorithm [14]; we call it sampling our neighbors' inverse neighbors. To motivate this technique, consider again Figure 11. Although the two remote nodes are unlikely to be neighbors of many other nodes, given that their existing neighbors are mostly nearby, they are likely to choose the same neighbors from outside their isolated domain. For this reason, they are likely to find each other in the set of their neighbors' inverse neighbors.

Normally, a DHT node would not record the set of nodes that use it as a neighbor. Actively managing such a list, in fact, requires additional probing bandwidth. Currently, the Bamboo implementation does actively manage this set, but it could be easily approximated at each node by keeping track of the set of nodes which have sent it liveness probes in the last minute or so. We plan to implement this optimization in our future work.

Recursive sampling Consider Figure 11 one final time, and assume that we are using a single-bit digits and that the two remote nodes begin with different digits, i.e. 0 and 1 respectively. The node whose identifier starts with 0 will have only one neighbor whose identifier begins with 1 (its level-0 neighbor). Likewise, the node whose identifier starts with 1 will have only one neighbor that starts with 0. The set of neighbors in whose inverse neighbor sets the two isolated neighbors can find each other is thus very small. As such, until the two isolated nodes have found

```
(1) function nearest_neighbors () =
(2)     $S$ = highest_nonempty_rt_level ();
(3)     $l$ = longest_matching_prefix ($S$);
(4)     while $l >= 0$
(5)         forall $n$ in $S$
(6)             $T$ = $n$.get_inverse_rt_neighbors ($l$);
(7)             $S$ = closest ($k$, $S \cup T$);
```

Figure 12: *The Tapestry nearest neighbor algorithm.*

very nearby level-0 neighbors, they will be unlikely to find each other among their neighbors' inverse neighbors.

To remedy this final problem, we can perform the sampling of nodes in a manner similar to that used by the Tapestry nearest neighbor algorithm (and the Pastry optimized join algorithm). Pseudo-code for this technique is shown in Figure 12. Starting with the highest level l in its routing table, a node contacts the neighbors at that level and retrieves their neighbors (or inverse neighbors). The latency to each newly discovered nodes is measured, and all but the k closest are discarded. The node then decrements l and retrieves the level-l neighbors from each non-discarded node. This process is repeated until $l < 0$. Along the way, each discovered neighbor is considered as a candidate for use in the routing table. To keep the cost of this algorithm low, we limit it to having at most three outstanding messages (neighbor requests or latency probes) at any time.

Note that although this process starts by sampling from the routing table, the set of nodes on which it recurses is not constrained by the prefix-matching structure of that table. As such, it does not suffer from the small rendezvous set problem discussed above. In fact, under certain network assumptions, it has been proved that this process finds a node's nearest neighbor in the underlying network.

4.3.3 Results

In order to compare the techniques described above, it is important to consider not only effective they are at finding nearby neighbors, but also at what bandwidth cost they do so. For example, global sampling at a high enough rate relative to the churn rate would achieve perfect proximity, but at the cost of a very large number of lookups and latency probes. To make this comparison, then, we ran each algorithm (and some combinations of them) at various periods, then plotted the mean lookup latency under churn versus bandwidth used. The results for median session times of 47 minutes are shown in Figure 13, which is split into two graphs for clarity.

Figure 13(a) shows several interesting results. First, we note that only a little bit of global sampling is necessary to produce a drastic improvement in latency versus the

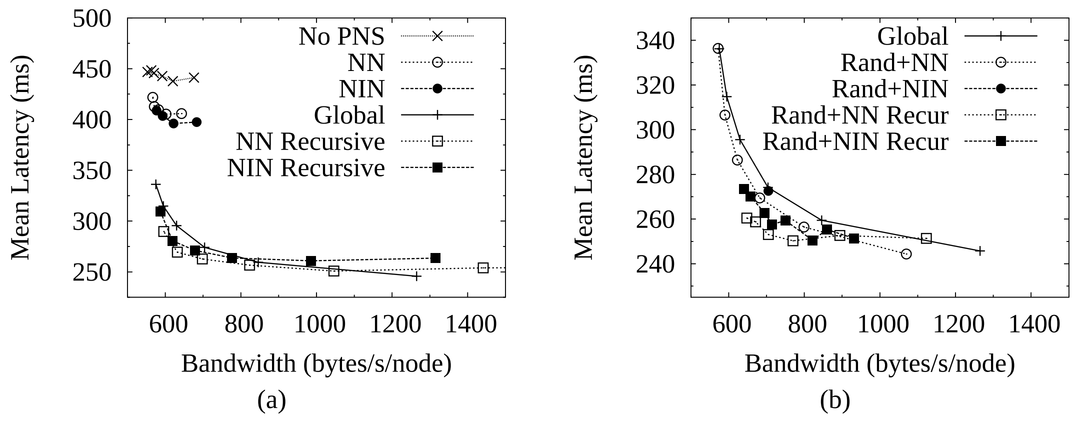


Figure 13: *Comparison of PNS techniques.* "No PNS" is the control case, where proximity is ignored. "Global Sampling" uses the lookup function to sample all nodes in the DHT. "NN" is sampling our neighbor's neighbors, and "NIN" is sampling their inverse neighbors. The recursive versions of "NN" and "NIN" mimic the nearest-neighbor algorithms of Pastry and Tapestry, respectively. Note that the scales are different between the two figures.

configuration that is not using PNS. With virtually no increase in bandwidth, global sampling drops the mean latency from 450 ms to 340 ms.

Next, much to our surprise, we find that simple sampling of our neighbor's neighbors or inverse neighbors is not terribly effective. As we argued above, this result may be in part due to the constraints of the routing table, but we did not expect the effect to be so dramatic. On the other hand, the recursive versions of both algorithms are at least as effective as global sampling, but not much more so. This result agrees with the contention of Gummadi et al. that only a small amount of global sampling is necessary to achieve near-optimal PNS.

Figure 13(b) shows several combinations of the various algorithms. Global sampling plus sampling of neighbors' neighbors—the combination used in our earlier work [23]—does well, offering a small decrease in latency without much additional bandwidth. However, the other combinations offer similar results. At this point, it seems prudent to say that the most effective technique is to combine global sampling with any other technique. While there may be other differences between the techniques not revealed by this analysis, we see no clear reason to prefer one over another as yet.

5 Related Work

As we noted at the start of this paper, while DHTs have been the subject of much research in the last 4 years or so, there have been few studies of the resilience of real implementations at scale, perhaps because of the difficulty of deploying, instrumenting, and creating workloads for such deployments. However, there has been a substantial amount of theoretical and simulation-based work.

Gummadi et al. [12] present a comprehensive analysis of the static resilience of the various DHT geometries. As we have argued earlier in this work, static resilience is an important first step in a DHT's ability to handle failures in general and churn in particular.

Liben-Nowell et al. [17] present a theoretical analysis of structured peer-to-peer overlays from the point of view of churn as a continuous process. They prove a lower bound on the maintenance traffic needed to keep such networks consistent under churn, and show that Chord's algorithms are within a logarithmic factor of this bound. This paper, in contrast, has focused more on the systems issues that arise in handling churn in a DHT. For example, we have observed what they call "false suspicions of failure", the appearance that a functioning node has failed, and shown how reactive failure recovery can exacerbate such conditions.

Mahajan et al. [19] present a simulation-based analysis of Pastry in which they study the probability that a DHT node will forward a lookup message to a failed node as a function of the rate of maintenance traffic. They also present an algorithm for automatically tuning the maintenance rate for a given failure rate. Since this algorithm increases the rate of maintenance traffic in response to losses, we are concerned that it may cause positive feedback cycles like those we have observed in reactive recovery. Moreover, we believe their failure model is pessimistic, as they do not consider hop-by-hop retransmissions of lookup messages. By acknowledging lookup messages on each hop, a DHT can route around failed nodes in the middle of a lookup path, and in this work we have shown that good timeout values can be computed to minimize the cost of such retransmissions.

Castro et al. [7] presented a number of optimizations they have performed in MSPastry, the Microsoft Research implementation of Pastry, using simulations. Also, Li et al. [16] performed a detailed simulation-based analysis of several different DHTs under churn, varying their parameters to explore the latency-bandwidth tradeoffs presented. It was their work that inspired our analysis of different PNS techniques.

As opposed to the emulated network used in this study, simulations do not usually consider such network issues as queuing, packet loss, etc. By not doing so, they either simulate far larger networks than we have studied here as in [7, 19], or they are able to explore a far larger space of possible DHT configurations as in [16]. On the other hand, they do not reveal subtle issues in DHT design, such as the tradeoffs between reactive and periodic recovery. Also, they do not reveal the interactions of lookup traffic and maintenance traffic in competing for network bandwidth. We are interested in whether a useful middle ground exists between these approaches.

Finally, a number of useful features for handling churn have been proposed, but are not implemented by Bamboo. For example, Kademlia [20] maintains several neighbors for each routing table entry, ordered by the length of time they have been neighbors. Newer nodes replace existing neighbors only after failure of the latter. This design decision is aimed at mitigating the effects of the high "infant mortality" observed in peer-to-peer networks.

Another approach to handling churn is to introduce a hierarchy into the system, through stable "superpeers" [2, 29]. While an explicit hierarchy is a viable strategy for handling load in some cases, this work has shown that a fully decentralized, non-hierarchical DHT can in fact handle high rates of churn at the routing layer.

6 Future Work

As discussed in the introduction, there are several other limitations of this study that we think provide for important future work. At an algorithmic level, we would like to study the effects of alternate routing table neighbors as in Kademlia and Tapestry. We would also like to continue our study of iterative versus recursive routing. As discussed by others [11], congestion control for iterative lookups is a challenging problem. We have implemented Chord's STP congestion control algorithm and are currently investigating its behavior under churn, but we do not yet have definitive results about its performance.

At a methodological level, we would like to broaden our study to include better models of network topology and churn. We have so far used only a single network topology in our work, and so our results should be not be taken as the last word on PNS. In particular, the distribution of internode latencies in our ModelNet topology is more Gaussian than the distribution of latencies measured on the Internet. Unfortunately for our purposes, these measured latency distributions do not include topology information, and thus cannot be used to simulate the kind of network cross traffic that we have found important in this study. The existence of better topologies would be most welcome.

In addition to more realistic network models, we would also like to include more realistic models of churn in our future work. One idea that was suggested to us by an anonymous reviewer was to scale traces of session times collected from deployed networks to produce a range of churn rates with a more realistic distribution. We would like to explore this approach. Nevertheless, we believe that the effects of the factors we have studied are dramatic enough that they will remain important even as our models improve.

Finally, in this work we have only shown the resistance of the Bamboo *routing* layer to churn, an important first step verifying that DHTs are ready to become the dominant building block for peer-to-peer systems, but a limited one. Clearly other issues remain. Security and possibly anonymity are two such issues, but we are unclear about how they relate to churn. We are currently studying the resilience to churn of the algorithms used by the DHT *storage* layer. We hope that the existence of a routing layer that is robust under churn will provide a useful substrate on which these remaining issues may be studied.

7 Conclusion

In this work we have summarized the rates of churn observed in deployed peer-to-peer systems and shown that existing DHTs exhibit less than desirable performance at the higher end of these churn rates. We have presented Bamboo and explored various design tradeoffs and their effects on its ability to handle churn.

The design tradeoffs we studied in this work fall into three broad categories: reactive versus periodic recovery from neighbor failure, the calculation of timeouts on lookup messages, and proximity neighbor selection. We have presented the danger of positive feedback cycles in reactive recovery and discussed two ways to break such cycles. First, we can make the DHT much more cautious about declaring neighbors failed, in order to limit the possibility that we will be tricked into recovering a non-faulty node by network congestion. Second, we presented the technique of periodic recovery. Finally, we demonstrated that reactive recovery is less efficient than periodic recovery under reasonable churn rates when leaf sets are large, as they would be in a large system.

With respect to timeout calculation, we have shown that TCP-style timeout calculation performs best, but argued that it is only appropriate for lookups performed recursively. It has long been known that recursive routing provides lower latency lookups than iterative, but this result presents a further argument for recursive routing where the lowest latency is important. However, we have also shown that while they are not as effective as TCP-style timeouts, timeouts based on virtual coordinates are quite reasonable under moderate rates of churn. This result indicates that at least with respect to timeouts, iterative routing should not be infeasible under moderate churn.

Concerning proximity neighbor selection, we have shown that global sampling can provide a 24% reduction in latency for virtually no increase in bandwidth used. By using an additional 40% more bandwidth, a 42% decrease in latency can be achieved. Other techniques are also effective, especially our adaptations of the Pastry and Tapestry nearest-neighbor algorithms, but not much more so than simple global sampling. Merely sampling our neighbors' neighbors or inverse neighbors is not very effective in comparison. Some combination of global sampling an any of the other techniques seems to provide the best performance at the least cost.

8 Acknowledgments

We would like to thank a number of people for their help with this work. Our shepherd, Atul Adya, and the anonymous reviewers all provided valuable comments and guidance. Frank Dabek helped us tune our Vivaldi implementation, and he and Emil Sit helped us get Chord up and running. Likewise, Peter Druschel provided valuable debugging insight for FreePastry. David Becker helped us with ModelNet. Sylvia Ratnasamy, Scott Shenker, and Ion Stoica provided valuable guidance at several stages of this paper's development.

References

[1] Freepastry 1.3. `http://www.cs.rice.edu/CS/Systems/Pastry/`.

[2] Gnutella. `http://www.gnutella.com/`.

[3] Inet topology generator. `http://topology.eecs.umich.edu/inet/`.

[4] MIT Chord. `http://www.pdos.lcs.mit.edu/chord/`.

[5] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proc. IPTPS*, Feb. 2003.

[6] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. 2003.

[7] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft, 2003.

[8] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. Apr. 2003.

[9] J. Chu, K. Labonte, and B. N. Levine. Availability and locality measurements of peer-to-peer file systems. In *Proc. of ITCom: Scalability and Traffic Control in IP Networks*, July 2002.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, Oct. 2001.

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

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

[13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. ACM SOSP*, Oct. 2003.

[14] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. SPAA*, 2002.

[15] V. Jacobson and M. J. Karels. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.

[16] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. IPTPS*, 2004.

[17] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proc. ACM PODC*, July 2002.

[18] B. T. Loo, R. Huebsch, I. Stoica, and J. Hellerstein. The case for a hybrid P2P search infrastructure. In *Proc. IPTPS*, 2004.

[19] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. IPTPS*, Feb. 2003.

[20] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS*, 2002.

[21] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA*, June 1997.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, Aug. 2001.

[23] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. Technical Report UCB//CSD-03-1299, University of California, Berkeley, December 2003.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, Nov. 2001.

[25] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. MMCN*, Jan. 2002.

[26] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proc. of ACM SIGCOMM Internet Measurement Workshop*, Nov. 2002.

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

[28] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI*, Dec. 2002.

[29] B. Y. Zhao, Y. Duan, L. Huang, A. D. Joseph, and J. D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proc. IPTPS*, March 2002.

[30] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1):41–53, Jan. 2004.