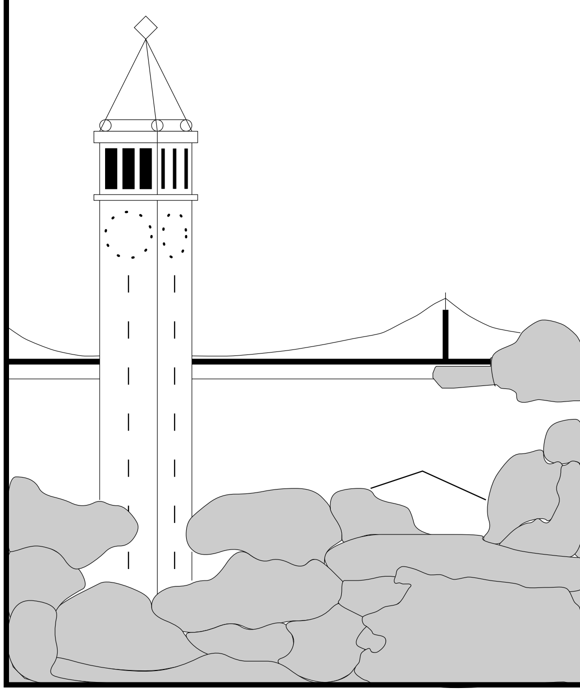
Handling Churn in a DHT

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



Report No. UCB/CSD-03-1299

December 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

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

December 2003

Abstract

This paper addresses the problem of *churn*—the continuous process of node arrival and departure—in distributed hash tables (DHTs). We demonstrate through experiment that existing DHT implementations break down at churn levels observed in deployed peer-to-peer systems, contrary to simulation-based results. We present Bamboo, a DHT that handles high levels of churn, and discuss the manner in which it does so. We show that Bamboo is able to function effectively for median node session times as short as 1.4 minutes, while using less than 900 bytes/s/node of maintenance bandwidth in a 1000-node system. This churn rate is faster than that observed in real file-sharing systems such as Gnutella, Kazaa, Napster, and Overnet. Since Bamboo's bandwidth usage scales logarithmically in the number of nodes, we expect this cost to remain within the reach of dialup modems even for very large systems. Moreover, in simulated networks without churn, Bamboo achieves lookup performance comparable with Pastry, an existing DHT with a similar structure.

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. [16, 19, 20, 23, 27]), which here 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 others have explored the tradeoffs in storing more or less state.

A second line of research into peer-to-peer systems has focused on observing deployed networks (e.g. [3, 7, 11, 21]). An important result of this research is that such networks are characterized by a high degree of churn, generally defined as the rate at which nodes join and leave the system. One important measure of churn is node *session time*: the time between when a node joins the network until the next time it leaves. Median session times as short as a few minutes have been observed in deployed networks.

This paper makes two primary contributions. First, we survey published studies of deployed peer-to-peer networks to derive requirements on the churn rates that DHTs must handle if they are to replace current systems. We then perform an empirical evaluation of the routing layers of existing DHT implementations, and we show that these implementations are unable to withstand the short session times observed in the wild. Beyond a certain level of churn, lookups in existing systems either take excessively long to complete, fail to complete altogether, or return inconsistent results. In addition, the ability of new nodes to join the DHT is often impaired.

Second, we describe Bamboo, a DHT that performs well under high levels of churn. Bamboo achieves this goal through the following three features of its design:

1. Static resilience to failures
2. Timely, accurate failure detection
3. Congestion-aware recovery mechanisms

Static resilience means that Bamboo can continue to perform lookups after node failures, routing around them even before recovery begins. To do so, however, it is critical that the system accurately distinguish down nodes from those with high loads or those across congested network paths. Failing to notice failures quickly leads to excessive lookup latencies, while assuming failure too soon leads to congestion collapse. A combination of active probing and recursive routing allows Bamboo to efficiently make this distinction. Finally, Bamboo integrates new nodes and recovers from the failure of old ones in a congestion-aware manner. Proactive recovery—where a DHT tries to react immediately to membership changes—only adds additional stress to an already-stressed network.

To avoid congestion collapse, Bamboo uses periodic algorithms, scaling back maintenance periods automatically in response to congestion.

This paper illustrates the importance of empirical testing of real implementations, particularly with regard to points 2 and 3 above. Existing studies of churn in DHTs have used simulations that 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, such network realities will exacerbate the ability of DHTs to handle even lower levels of churn.

We compare Bamboo with two popular DHT implementations for which working source code is available.[1] It is possible that there exist DHT implementations with churn resilience comparable to or better than Bamboo, but we are unaware of any published work that demonstrates their performance or discusses the techniques they use. Furthermore, while we show that other DHTs suffer under moderate levels of churn, we cannot pretend to be experts on their implementation and so can only speculate on why they do so. Instead, we present results for these DHTs because they were useful to us in gaining insights into the problem when developing Bamboo. Our claim is simply to have designed and built one DHT which can handle churn well, using techniques which can be applied to other, similar systems.

The rest of this paper is structured as follows: in Section 2 we examine churn rates observed in peer-to-peer file-sharing networks, derive robustness requirements for a DHT in such an environment, and show the behavior of an existing DHT implementation under these conditions. In Section 3 we introduce Bamboo, focusing on its hybrid geometry and datagram networking layer, and in Section 4 we discuss in detail how Bamboo remains stable and available under high rates of node churn. In Sections 5 we describe the experiments we performed to validate the design of Bamboo and present the results. In Section 6 we survey related work, and we conclude in Section 7.

2 Churn

What kinds of churn must a DHT expect in a real application? To answer this question, we surveyed published empirical studies of peer-to-peer file sharing networks. With the exception of Overnet [2], which uses Kademlia [16], the networks measured do not employ DHTs. Nevertheless, the behavior seen in these networks is a useful guide to what a robust DHT should handle. We first present some definitions to help clarify the space.

2.1 Definitions

As illustrated in Figure 1, *session time* is the time between when a node joins a network until it subsequently leaves the network. In contrast, a node's *lifetime* is the time from when it enters the network for the first time until the time at which it leaves the network permanently. Finally, a node's *availability* is often defined as the sum of its session times divided by its lifetime.

The length of lifetimes in a system mostly affects long-term application-level durability. For example, in a peer-to-peer storage system, long lifetimes are a prerequisite; without them, the maintenance traffic required to restore data replication levels as hosts leave the system exceeds available host bandwidth [4]. In contrast, a file sharing application such as Overnet only stores the locations of files, rather than the files themselves, in the DHT. As such, there is little information to restore when a host leaves the network, allowing the application to withstand short lifetimes. Other peer-to-peer applications also have modest storage requirements; for example, an instant messaging application could use a DHT as its rendezvous service. We might expect such an application to experience short session times, too; like file sharing, when a user is not actively using the system, there is little motivation to pay the bandwidth costs associated with participating.

Regardless of the application layer of a DHT, the routing layer is mostly sensitive to the length of session times. Even temporary loss of a routing neighbor weakens the correctness and performance guarantees of a DHT. Unavailable neighbors reduce a node's effective connectivity, forcing it to choose suboptimal routes and increasing the destructive potential of future failures. Finding a replacement for the lost neighbor is a priority, and this neighbor maintenance makes up a significant portion of the bandwidth consumed.

Since all applications depend on the routing layer, in this work we focus on the effects of short session times. We leave analysis of the effects of short lifetimes on other layers as important future work.

2.2 Empirical studies

We use empirical studies of file-sharing networks to derive requirements on handling churn for two reasons. First, almost all deployed peer-to-peer systems of any size today for which measurements are available are file-sharing applications. Second, we are interested in the natural question of whether DHTs can replace the unstructured overlays in existing file-sharing applications.

[1] Lamentably, we could find no freely-available, complete DHT that used more than $O(\log N)$ state in return for shorter lookup paths.

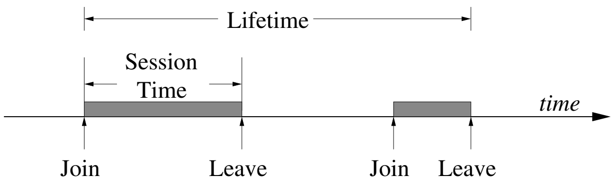


Figure 1: *Metrics of churn.* With respect to the routing and lookup functionality of a DHT, the *session times* of nodes are more relevant than their *lifetimes*.

First Author	Systems Observed	Session Time
Saroiu [21]	Gnutella, Napster	50% $\leq$ 60 min.
Chu [7]	Gnutella, Napster	31% $\leq$ 10 min.
Sen [22]	FastTrack	50% $\leq$ 1 min.
Bhagwan [3]	Overnet*	50% $\leq$ 60 min.
Gummadi [11]	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.

Saroiu, Gummadi, and Gribble [21] presented the earliest study we have found of session times in peer-to-peer systems. Using active probing, they found the median session time in both Napster and Gnutella to be around 60 minutes. Another active study of Napster and Gnutella by Chu, Labonte, and Levine [7] found that 31% of observed sessions were shorter than 10 minutes, and less than 5% were longer than 60 minutes. On the other hand, they observed a small fraction of sessions (less than 0.01%) lasting thousands of minutes at a time.

Sen and Wang [22] used passive monitoring to observe FastTrack traffic using routers in an ISP backbone. To compute session length, they included all traffic less than 30 minutes apart from the same IP address, and found that 60% of nodes had a total session time of under 10 minutes daily.

Bhagwan, Savage, and Voelker [3] performed an active study of the Overnet system. The choice is significant since nodes in Overnet are uniquely identified by names that persist across sessions. As such, these names are more suitable for many metrics than IP addresses which vary over time due to DHCP, firewalls, etc. While this distinction is important for measuring node lifetimes, changing IP address involves leaving and rejoining a network, so we believe the previous studies' session time results are still valid.

Since the Overnet study did not include session times, we re-analyzed their data to extract them. This data contains the results of active probes for 2,400 distinct Overnet hosts every 20 minutes over a week. Marking the start of a session as the transition from a host being unreachable to being reachable, or as the change from one IP address

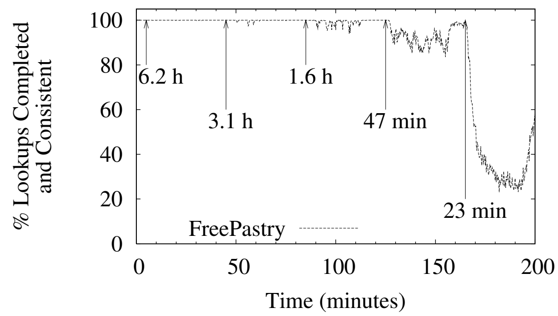


Figure 2: *FreePastry under churn.* Shown is the percentage of lookups that complete in a 1000-node FreePastry network under increasing levels of churn. Session times for each churn period are indicated by arrows.

to another, we found a median session time of 60 minutes, plus or minus the 20 minute probe period.

A study of Kazaa recently published by Gummadi et al. [11] used passive measurement techniques to estimate session times as the length of continuous periods during which a node was actively retrieving files. They found a median session length of only 2.4 minutes, and a 90th percentile session length of 28.25 minutes.

The session times observed in the works referenced above are summarized in Table 1. We conclude that a peer-to-peer network built on a DHT should be robust for session times from an hour down to a minute. Also, with the exception of Overnet, session times seem to have shortened over time. The relatively long length of Overnet sessions may be because it is primarily a movie-sharing service, and movies take longer to download than other files such as music.

Neither the studies we have cited nor our analysis take into account the possibility that sessions are cut short due to network failures, and that a robust DHT would experience longer session times due to its own resilience. Nevertheless, we feel that our derived requirements are a useful starting point for DHT designers.

2.3 Existing DHTs

Figure 2 shows one effect of churn on a robust, reasonably mature DHT: a 1000-node network of nodes running FreePastry on a cluster of 40 machines under ModelNet [24] to simulate realistic network latencies and bandwidths. FreePastry is an open source implementation of Pastry; in the remainder of this paper, we use "FreePastry" to refer to this particular implementation, and we use "Pastry" to refer to the Pastry algorithm in general. The graph shows the percentage of key lookups which successfully return correct results against time. We create node churn during the experiment over 30 minute periods, separated by calm periods long enough for the DHT to recover (10 minutes in this case). The churn rate is increased for each period.

The graph in Figure 2 is shown with time on the x axis to illustrate the nature of the tests we use in this paper. Later, in Section 5, we will transform this graph to show the metrics as a function of the churn rate, and we will add numbers for Chord and Bamboo. We will show that both Chord and FreePastry become unusable at the higher levels of churn observed in real peer-to-peer networks, although for different reasons.

We make a final comment on this graph. Between all but the last two churn periods, FreePastry recovers completely, once again correctly completing all lookups. Both Chord and FreePastry exhibit this behavior; so long as a sufficient quiet period follows, they are able to withstand rather significant network perturbations. The difficulty with churn is that there is no such quiet period; the network is in a continual state of recovery.

3 Bamboo Preliminaries

Bamboo is the DHT we have been building since January, 2003. It consists of just over 10,000 semicolons of Java, of which about 3,300 make up the routing layer studied in this paper. Before we discuss the merits of our design with respect to handling churn, we do two things to set the stage. First, we present the Bamboo geometry and routing algorithm. As defined by Gummadi et al. [10], the *geometry* of a DHT is the pattern of neighbor links in the overlay network, independent of the routing algorithms or state management algorithms used. Second, we present the congestion-controlled networking layer which Bamboo uses to communicate between nodes, since features of its design are integral to the algorithms we discuss later. In Section 4, we describe how Bamboo acquires and maintains routing state, and how it uses that state to route messages.

3.1 Network geometry

The Bamboo geometry is identical to that of Pastry, which we briefly summarize here, following the notation of Rowstron and Druschel [20] for consistency. We refer the reader to that work for more details.

Each node in Bamboo is assigned a numeric identifier from the range $[0, 2^{160})$, 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. Each node in the Bamboo network maintains a *leaf set*—the set of $2k$ nodes immediately preceding and following it in the circular identifier space. We denote this set by L,

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 3: *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.

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 addition to its leaf set, each Bamboo node maintains a *routing table*. 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 $(l + 1)$th digit is i. Like Pastry, Bamboo tries to choose the node closest to it in network latency from all nodes that can fill each routing table entry.

Algorithmically, routing in Bamboo proceeds as shown in Figure 3. 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.

We have described this routing in *recursive* terms (where a message is forwarded by a series of nodes en route to its destination), but it is of course possible to route *iteratively*, where the originating node performs a series of lookups to intermediate nodes before sending the message directly to the destination [23]. However, the choice of recursive routing in Bamboo turns out to be important in handling churn, and we discuss it further in Section 4.2.

As discussed in [20], this routing table design performs lookups in $O(\log N)$ hops, 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 Bamboo geometry; Gummadi et al. [10] 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. Such static resilience is clearly useful in handling

```
public static interface SendCallBack {
    void send_callback (Object user_data, boolean success);
}

public void send (
    Object msg, InetSocketAddress dst,
    int tries, SendCallBack cb, Object user_data);

public double est_rtt_ms (InetSocketAddress peer);
```

Figure 4: *The Bamboo communications layer interface.* The layer makes up to *tries* attempts to send *msg* to *dst*, calling *send_callback* after an ACK or too many retries. It also exposes the mean round-trip time to each peer.

failures in general and churn in particular, and it was the reason we chose the Pastry geometry for use in Bamboo.[2]

3.2 Communications layer

Bamboo nodes communicate using UDP. Originally, we chose UDP to avoid the difficulties we had encountered with file descriptor limits while running many virtual DHT instances over TCP on the same physical machine; such virtualization is common in testing large networks on limited physical resources and is quite useful for debugging. Since that time, as explained in Section 4.2, we have come to believe that the semantics of TCP are inappropriate for a DHT; instead, what is needed is message-based, unreliable, unordered, but congestion-controlled communication. The manner in which these semantics are provided is briefly described below, but we emphasize here that it should be viewed only as an artifact of the system.[3]

In the style of TCP, the Bamboo communications layer uses the time between when it sends message and the receipt of the corresponding ACK to maintain an exponentially weighted average round trip time (RTT) and variance thereof for each peer. These values are made available to higher layers of the system. It computes round-trip timeouts (RTOs) to decide when to retransmit a packet, and it backs the RTO off exponentially with each timeout. It maintains a congestion window in a similar manner to the TCP slow-start algorithm, and it notifies the application when a packet is acknowledged. The interface that the communications layer exports is shown in Figure 4.

[2]We could also have used a pure ring geometry as in Chord, extending it to account for proximity in neighbor selection as described in [10].

[3]In fact, the semantics we desire are quite close to those provided by DCCP [12] using TCP-like congestion control, and it is likely that we would have used DCCP were it available, although we admit we have not fully explored this possibility.

function join (A, G) =
  G' = nearest_neighbor (A, G);
  B, P = lookup (G', ID_A);
  L = get_leaf_set (A);
  for i **from** 0 **to** $|P| - 1$
    l = length_of_longest_matching_prefix ($\mathrm{ID}_A, \mathrm{ID}_{P_i}$);
    R_i = P_i.get_routing_table_level (l)

Figure 5: *The Pastry join algorithm.* The new node is A, and its given gateway is G.

4 Bamboo under churn

Bamboo exhibits robustness under churn through the combination of three key features: static resilience to failures, timely, accurate failure detection, and congestion-aware recovery mechanisms. Static resilience provides routability after failure even before recovery takes place, and so allows the DHT to use the power of its own routing mechanism to enact that recovery. Below, we describe the latter two features listed above and how they improve Bamboo's resilience to churn.

4.1 Congestion-aware recovery

Construction of a Bamboo network and the maintenance thereof use the same algorithms; both the failure of an existing node and the appearance of a new one are viewed as disruptive events from which the network must recover. This decision yields economy of mechanism, simpler code, and most importantly helps the network deal with churn.

4.1.1 The Pastry join algorithm

Originally, Bamboo used the Pastry join algorithm (Figure 5). To motivate the new design, we first review Pastry's algorithm and the challenges of implementing it. To join an existing network a Pastry node A needs the IP address and port of another node G that has already joined; we call this node the *gateway*, or *bootstrap* node. A uses G to find a closer gateway G' and uses G' to lookup its own identifier ID_A to find node B, from which it gets its leaf set. It then builds its routing table by contacting the nodes on the lookup path P. Pastry's geometry ensures that the identifiers of the nodes along the lookup path for ID_A share successively longer prefixes with ID_A, making them appropriate candidates to fill A's routing table.

Castro et al. [5] define a *probe* as the process with which one node determines the latency to another node, independent of network congestion. They estimate that a Pastry node must perform an average of 26–31 probes to join the network and that an average of 50–70 existing nodes issue an average of one or two probes each in response to the join. While this number is not large, obtaining a good probe result is difficult. The quality of the probe matters because after the join, the neighbor choices are infrequently re-evaluated. In an uncongested network the minimum of a few ping times may suffice, but under congestion the situation is more complex since probes interfere with each other.

Besides active probing, there are other options for estimating or otherwise obtaining information on network latencies [6, 8, 13], but performing a direct comparison of these approaches with Bamboo's existing network measurement facility is a topic for future work.

4.1.2 Bootstrapping Bamboo

Rather than solve this network measurement problem, we decided to avoid it in Bamboo. A new Bamboo node performs only lines 1, 3, and 4 of the Pastry join algorithm in Figure 5, using G for G' on line 3. It adds the nodes in P to its routing table, but sends no latency probes. After this quick bootstrapping, it considers itself joined; its routing table is then filled and optimized through a continuous process that stops only when it leaves the network. Before describing this process in full, we first examine the benefits.

First, to route *efficiently*—in $O(\log N)$ hops—it suffices to fill each routing table entry with *any* node of the appropriate prefix. As we show, this can be done quickly and cheaply, and so we decouple it from the more expensive process of finding nearby neighbors. We say the latter process allows a node to route *quickly*, but its benefit relative to the gain obtained in going from leaf-set-only routing—which requires $O(N)$ hops—to efficient routing is not large, so it we give it lower priority. Second, since the choice of neighbor to fill any entry of the routing table is continually re-evaluated, the importance of any one measurement is less. An early mistake causes a suboptimal choice only until later measurements correct it. Finally, churn is the normal state of the network. Tuning the routing table at join time to include only the closest available neighbors is tuning to a state that is sub-optimal moments later. Under these conditions, we believe a continual optimization process is more appropriate.

In the following sections, we describe the components of this optimization process. We construct the process from short, simple actions because they are less likely to be interrupted by failures and more likely to return improvements relevant to the current state. In some cases we can show analytically that the system converges to a good state quickly; in others we study the convergence through experiment.

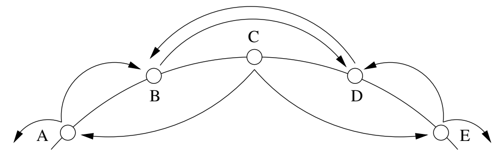


Figure 6: *The need for pushing and pulling leaf sets.* Arrows represent neighbor links. Unless leaf sets are also pulled, C's leaf set is never corrected.

4.1.3 Leaf set maintenance

Bamboo's leaf set is kept current by an epidemic algorithm [9]: every period a node sends its entire leaf set to a randomly-chosen neighbor in the set, which in turn replies with the contents of its leaf set.[4] We call the first message a leaf set *push* and the reply a leaf set *pull*. The information shared in this transaction is the set of nodes in a particular area of the circular name space. In the published descriptions of Pastry, nodes only push leaf sets; there does not appear to be a corresponding pull [15].

Pulling leaf sets results in greater resilience to arbitrary bad states. An example is shown in Figure 6; indeed, it was observing this kind of state which led us to implement pulls. Five nodes are shown in a system with $k = 1$; the arrows represent each node's successor and predecessor according to its leaf set. Node C is unavailable during which time B and D join. C subsequently becomes available again, but nodes B and D have no knowledge of it, whereas C still thinks its neighbors are A and E. If leaf sets are only pushed, no node in this system will tell C about the existence of B or D, and its leaf set will remain incorrect. With pulls, however, the first time C contacts A it will learn about B; the same is true for E and D.

In Bamboo, the bandwidth consumed by neighbors of a new node A is not increased by the join operation, apart from A's initial lookup. When A joins the DHT, it obtains an initial leaf set from the node closest to its identifier, but does not immediately announce its presence to other nodes. Instead, these nodes learn about A through normal leaf-set push and pull messages. When many nodes join simultaneously, this technique stresses the underlying physical network far less than if each joining node were to actively announce itself to all others. In other words, we decouple maintenance bandwidth from churn rate as much as possible, making it depend only on leaf-set size.

Furthermore, this technique does not affect the consistency of the ring for long: consider a new node A that joins a previously-consistent Bamboo network and receives its leaf set from a node B. Since B's leaf set was already consistent, A can construct a consistent leaf

[4] We explore the sensitivity of Bamboo's performance to the frequency of this and other periodic processes in Section 5.4.

```
function global_tuning (A, R, l, d) =
    I = random_identifier ();
    I' = ID_A[0, l − 1] + d + I[l + 1,|I| − 1];
    B = A.lookup (I');
    if R_l[d] = null or B.closer_than (R_l[d])
        R_l[d] := B;
```

Figure 7: *Global tuning.* To improve routing table entry $R_l[d]$, node A looks up an identifier with its first l digits, d for its $l+1$st digit, and otherwise all random digits. The returned node is used if it is closer than the existing entry.

```
function local_tuning (R, l) =
    d = random_digit ();
    L = R_l[d].get_routing_table_level (l);
    for i from 0 to |L| − 1
        if R_l[i] = null or L[i].closer_than (R_l[d])
        R_l[d] := L[i];
```

Figure 8: *Local tuning.* To improve routing table level R_l, a node choses a random entry of level R_l and gets level l of its routing table, then probes each returned entry to see if it is closer than the existing entry.

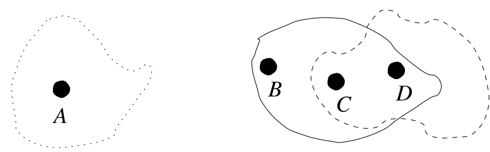


Figure 9: *A problem for local tuning.* If A joins using D as its gateway, its initial level-0 neighbors are the same as those of D; assume that these are all within the dashed line. In local tuning, A contacts a level-0 neighbor, e.g. C, and asks it for its level-0 neighbors. A would learn about B in this manner. However, there may be no path from the D's ideal neighbors to those of A.

set immediately. Likewise, B's leaf set is made consistent again by the inclusion of A. Now consider the k nodes in either side of A's new leaf set. Each node in this set sends each push to another node in the set with probability at least 1/2 (the other half of the time the push goes to a node outside the set), so the process of spreading knowledge of A behaves like a simple epidemic with period at most twice the push period. Such epidemics are known to "infect" all nodes in $O(\log k)$ time for a set of k nodes [17], so we can conclude that the leaf sets in the affected area of the ring will quickly return to consistency. Of course, under churn conditions this static argument does not hold, and the analysis of convergence times is more difficult, but our experiments in Section 5 show that it is still reasonable.

4.1.4 Routing table maintenance

Unlike leaf sets, routing tables in Bamboo are asymmetric; the nodes in the routing table for a node A do not necessarily have A in their routing tables. Moreover, there are generally many available neighbors for each entry in a A's routing table, and ideally A would choose the one closest in network latency. As such, a simple, periodic, pair-wise sharing of routing tables is insufficient to keep them well tuned, and more sophisticated algorithms are necessary. In Bamboo, we use two such algorithms, one called *global tuning* of routing tables, and the other called *local tuning*.

Global tuning. The global tuning algorithm will always bring a static network to an optimal state but is slow relative to local tuning. It has no equivalent in the Pastry algorithm, but it is similar in character to what Chord calls *stabilization*, using the fact that Bamboo can always route to the node whose identifier is closest to any destination identifier so long as all leaf sets are correct.

As shown in Figure 7, to fill or improve routing table entry $R_l[d]$, node A does a lookup on an identifier with its first l digits, d for its $l+1$st digit, and otherwise all random digits. The returned node is used if it is closer than the existing entry. For example, to fill a hole in level-1, digit-4 of its routing table, the node 0x123 might do a lookup on identifier 0x147. It can be shown that with probability $\geq 1/2$ this lookup will return some node that starts with 0x14 if one exists in the network, and it will return the closest such node with non-zero probability.

Local tuning. The local tuning algorithm is similar to Pastry's routing table maintenance, except that Bamboo performs it incrementally and more often. As shown in Figure 8, a node A periodically contacts a random member $R_l[d]$ of A's routing table and requests all of its entries from level l. If a returned entry fills a hole in A's routing table, it is used; the other entries are probed for latency before possibly replacing existing entries in A's routing table.

Local tuning returns many possible neighbors for each request, so it can fill most holes quickly. Moreover, it is useful for improving routing tables when most of the tables in the system are already near optimal. For example, say that node A has close neighbors for all digits on the 0th level of its routing table except for some digit d. If any of its existing neighbors at level-0 have a close neighbor starting with digit d, then (assuming the triangle inequality) if A performs local tuning with one of those neighbors, it will discover the neighbor starting with digit d and be able to improve its routing table.

However, local tuning is not sufficient to improve routing tables in all systems. A simple bad scenario is shown in Figure 9. A joins using node D as its gateway. Its first set of level-0 neighbors—obtained through local tuning—are the same as those of D; assume that these all fit into the (physical network) area bounded by the dashed line, and that A's ideal level-0 neighbors all fit within the dotted line. In local tuning, A contacts an existing level-zero neighbor, e.g. C, and asks for all of its level-zero neighbors. In this example, A would learn about B in this manner. Since there may be no path through these level-0 neighbor links from D's ideal neighbors to those of A, A needs a way to jump across gaps in this graph of level-0 neighbor links.

Global tuning helps, eventually picking a node on A's side of the gap. However, in the worst case, if A lies close to some small number of other nodes, all of which are widely separated from the remainder of the network, and node A is initially unaware of these physically close nodes, it is unlikely that it will learn about them in a timely manner. We quantify the extent of this problem through simulation in Section 5, but finding a new algorithm which helps A discover these close neighbors more quickly—while sharing the continual-optimization style of global and local tuning—is an important subject for future work.

4.1.5 Reacting to congestion

Each Bamboo node keeps track of whether it has a leaf set push-pull, global routing table tuning request, or local routing table tuning request in progress at any given time, and it does not start a subsequent operation until it has completed the previous one of the same type or until it has decided the associated neighbor has left the network. This means that a given set of periods set the maximum bandwidth consumed by the node for maintenance, but the actual bandwidth consumed may be lower in the presence of congestion. In our future work, we would like to explore combining our techniques with automatic setting of periods as by Mahajan, Castro, and Rowstron [15]. In the meantime, we study Bamboo's sensitivity to the setting of these periods in Section 5.4.

4.2 Timely, accurate failure detection

The routing algorithm itself is only one part of how routing is actually performed in practice; we must also specify what to do in the case of failures. In a traditional client-server system, such as NFS, the server does not often fail, and when it does there are few options for recovery. If a response to an NFS request is not received in the expected time, the client must usually try again with an exponentially increasing timeout value.

In a peer-to-peer system, in contrast, churn is the norm, and quite often requests will be 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. Consequently, backing off the request period is not only an insufficient solution for handling request timeouts, but a node often has the opportunity to immediately retry a request through a different neighbor in the event of a timeout.

Before routing to a different neighbor, a node must ensure that the timeout for the first request was judiciously selected. 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 into the system may cause further dropped packets, resulting in more requests being retransmitted, eventually leading to congestion collapse. 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, 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. Bamboo chooses such timeouts through two complementary techniques. First, it performs active probing through the user-level networking layer. As discussed above, this layer maintains an exponentially weighted mean and variance of the response time for each neighbor with which the local node communicates. When actual traffic—lookup requests, leaf set changes, etc.—is being sent to some neighbor, that traffic is used to maintain this timing information; in its absence, dummy requests are sent (every 4 seconds by default). As a result, a Bamboo node always has a recent estimate of the response time for each of its neighbors.

Of course, it is infeasible for every node in the DHT to actively probe every other node at scale. This dilemma is overcome by Bamboo's use of recursive, rather than iterative, routing. In recursive routing, a lookup request is forwarded from node to node until it reaches its destination. In contrast, in iterative routing, the querying node contacts each intermediate node directly. Since a node has only $O(\log N)$ neighbors, the use of recursive routing allows a Bamboo node to only communicate with a small number of peers, and active probing is feasible.

In contrast, with iterative routing, a node must potentially have a good timeout for *any* other node in the network, and it is not yet clear how good estimates can be obtained. One approach is the use of a synthetic coordinate system as in Chord [6, 8]; this algorithm uses machine learning to predict the latency to arbitrary hosts though measured distances to other hosts. A concern with this approach is that the round-trip delay to a host during periods of high network congestion or end-host load can be significantly longer than the usual delay. If such periods are short relative to the convergence time of the estimator, there is a danger of spurious retransmissions during these high congestion periods—precisely when retransmissions are most disruptive to the system. To our knowledge, the temporal estimation error of these algorithms has not been studied; such a study would be valuable to DHT implementations using iterative routing.

4.3 Summary

Before continuing to the experimental portion of this paper, we take a minute to review the key features of our design. First, Bamboo nodes acquire and maintain neighbor state by several congestion-aware, continuous optimization processes. This process is divided into separate concerns, first producing correctness through maintenance of the leaf sets, then producing efficiency by filling the routing table, and finally tuning routing table entries for proximity. In all cases, the rate at which these messages are sent is capped and is further scaled back in response to congestion or load. Second, Bamboo nodes actively probe their neighbors to compute good timeout values, and by using recursive routing they avoid sending messages to nodes for which they do not have such timeouts.

5 Results

In this section we demonstrate that Bamboo can in fact handle churn for session times as short as a few minutes and show results for other DHTs for comparison. We then examine the design factors responsible for this performance, show Bamboo's sensitivity to the length of its computed maintenance periods, and examine the costs our design incurs over other DHTs when run on churn-free networks.

5.1 Experimental setup

Our platform for measuring DHT performance under churn is a cluster of 40 IBM xSeries PCs 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 [24] to impose wide-area delay and bandwidth restrictions, and the Inet topology generator[5] 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.

We measure Bamboo, FreePastry[6] release 1.3, and a recent CVS snapshot (8/4/2003) of Chord[7]. Chord is run 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. FreePastry is run using the default 24-node leaf sets and a logarithm base of 16. Bamboo is run with 8-node leaf sets, and with a logarithm base of either 2 or 16, called "base 2" or "base 16" mode, respectively, in the results.

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.[8] 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. Nodes may issue multiple outstanding lookups, and our control wrapper does not timeout, cancel or retry any lookups.

We then repeatedly churn nodes until the system performance stabilizes; this phase normally lasts 20-30 minutes, but can take an hour or more. A new node is started each time a node is killed, maintaining the total network size at 1000. Node death events are chosen by a Poisson process, and are therefore uncorrelated and bursty. This model of churn is similar to that described by Liben-Nowell et al. [14]. We used churn rates of 8 deaths/second to 1 death/minute, corresponding to median session times of 1.4 minutes to 3 hours.

5.2 Bamboo under churn

Figure 10 shows the effects of churn on each DHT running under ModelNet. We focus on four areas—the number of started nodes which have successfully joined the network and begun issuing lookups, the percent of lookups that complete successfully, the consistency of completed

[5] `http://topology.eecs.umich.edu/inet/`

[6] `http://www.cs.rice.edu/CS/Systems/Pastry/`

[7] `http://www.pdos.lcs.mit.edu/chord/`

[8] We do not attempt to enforce proximity between a new node and its gateway, as suggested for best FreePastry performance; this decision only effects the proximity of a FreePastry node's neighbors, not the efficiency of its routing.

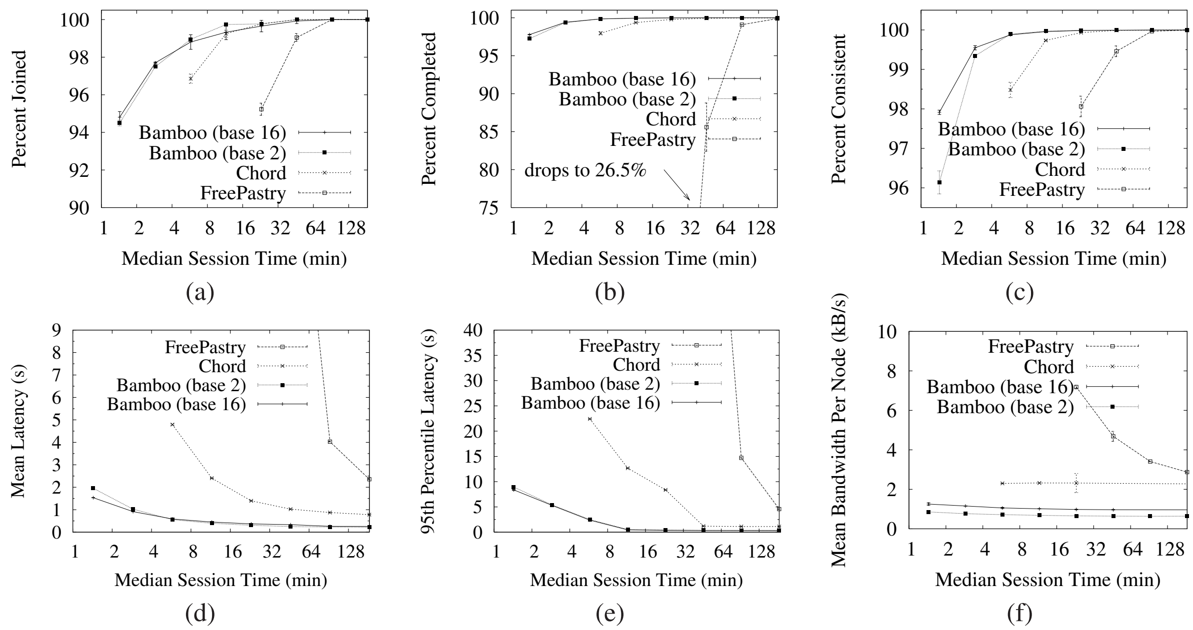


Figure 10: *Metrics of success.* The percentage of started nodes which successfully join (a), the percentage of lookups that return some (possibly incorrect) result (b), the percentage of completed lookups that are also consistent (c), the mean latency of completed lookups (d), the 95th percentile latency of completed lookups (e), and the bandwidth used per node (f). Each experiment was run three times. Error bars show mean and standard deviation across runs.

lookups, and the latency of completed lookups—all while varying the rate of churn. Under FreePastry and Bamboo, whether a node has joined the DHT is defined by whether the *join* function has returned. Under Chord, we say the network has joined when it has received its successor's successor list. This definition is the same as used by the DHASH layer build atop Chord. In all cases, the failure of a node to join is ignored if it is killed within two minutes of joining. Each point in Figure 10 represents the mean value over at least three trials; the error bars represent one standard deviation.

A lookup is said to complete if any result is returned. To measure lookup consistency, each lookup is simultaneously performed by ten different nodes in the network and the results are compared (since our tests run on a cluster, the machines involved have closely synchronized clocks). If there is a majority among the results, any result not in the majority is considered an inconsistency; if there is no majority, all results are considered inconsistent. For some applications (e.g. DHASH) this notion of consistency is more strict then necessary, and we hope to investigate other, less strict definitions in the future. The latency of a lookup is simply the time from which it is issued until some result is returned.

Bamboo handles all the churn rates shown quite well. Its percentage of nodes joined never falls below 94%, it percentage of lookups completed never falls below 97%, it percentage of lookups that are consistent never falls below 95%, and its 95th percentile latency never exceeds 9 seconds. At the moment, our testing code is unable to produce median session times much shorter than 1.5 minutes, but we are working to extend it in order to find the point at which Bamboo collapses under the load.

For comparison, we note that Chord shows similar curves to Bamboo for percentage joined, completed, and consistent, except that they fall off a factor of 4 sooner. More interestingly, Chord's latency seems to fall off far more quickly with increasing churn than Bamboo's. With respect to latency, Chord and Bamboo differ in two key respects: proximity in neighbor selection, and iterative vs. recursive routing. The proximity of Bamboo's neighbors only gets worse with increasing churn, as there is less time to optimize the routing table. Algorithmically, recursive routing should be no more than a constant factor faster than iterative. Instead, we believe the divergence in latency occurs because of the difficultly in computing accurate message timeouts in iterative routing; we explore this hypothesis in Section 5.3.2.

Finally, it is clear that FreePastry can handle churn with median session times longer than 90 minutes, but that around 45 minutes it begins to suffer, and for median session times of 23 minutes or less it has completely collapsed, completing less than 30% of the lookups issued to it. It is also clear that FreePastry is significantly slower than Chord or Bamboo, but we believe that it is so because it uses Java RMI for communication between nodes, whereas the other two DHTs use custom messaging layers. We believe Pastry's use of proactive recovery is the cause of its difficulty with high rates of churn; we explore this hypothesis in Section 5.3.1.

5.3 Causes of failure

In this section we investigate some of the ways in which a DHT can respond to churn and demonstrate the importance of the decisions we have made in building Bamboo.

5.3.1 Proactive vs. periodic recovery

Since churn is a process of nodes joining and leaving a network, we investigate the reaction of each DHT to join and leave events. By doing so, we hope to gain insight into the reasons behind their performance under churn.

Figure 11.a shows the per-node bandwidth usage of 1000-node Chord, FreePastry, and Bamboo networks undergoing the simultaneous failure of 200 nodes. According [20], when a Pastry node A notices that one of its neighbors B has failed, it contacts another neighbor to retrieve a replacement for B. We call this type of recovery *proactive*, in that node A is proactively seeking a replacement for B as soon as it notices that B is down. In contrast, Chord and Bamboo perform *periodic* recovery; on a periodic basis, they "recover" a neighbor whether it is down or not. In Chord, this process is called the *stabilization* of that neighbor, whereas in Bamboo it is the leaf set and routing table maintenance algorithms described in Section 4.

In the sense that a join represents a modification of the network, we can also speak of the process of integrating a new node as a recovery—a movement of the network from a bad state to a good one. Figure 11.b shows the per-node bandwidth usage as 200 new nodes join a network of 1000 existing ones. The difference between FreePastry and the other two networks is similar to the failure case. As discussed in Section 4.1.1, on a join, a Pastry node tries to immediately create a good routing table, asking other nodes for their routing state, pinging several nodes for proximity, etc. Moreover, existing Pastry nodes ping the new node. Once again, we call this process a proactive approach. In contrast to Pastry, both Chord and Bamboo undergo a more periodic approach to integrating themselves into the network. Using the same periodic processes used to recover from failures, the nodes in these DHTs periodically discover new, more appropriate neighbors.

The distinction between these two methods of recovery is important because the appearance of a node being down and the bandwidth used by the network are not independent. If the failure of a node causes the network to use more bandwidth in attempt to recover, some packets may be dropped. If those drops cause timeouts which are interpreted as other failures, a positive feedback loop is formed and the network may collapse. Initially, Bamboo was implemented to recover from joins and leaves in the same manner as Pastry, and it was our observation of such a positive feedback loop (and our experience testing the Chord implementation) that lead us to redesign the Bamboo neighbor maintenance algorithms in the first place.

At low rates of churn, or where bandwidth is plentiful, proactive recovery brings a network back into consistency more quickly than periodic recovery. However, Figure 10 shows that Bamboo's percentage of lookups completed and consistent are quite high for low rates of churn. Since its periodic recovery style is also effective at high rates of churn, we see it as a clear advantage over proactive recovery.

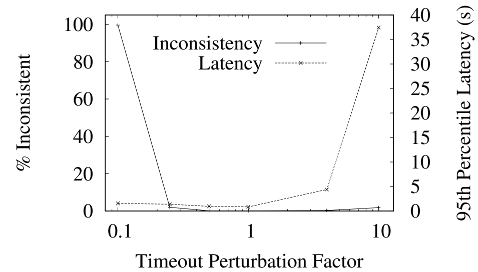


Figure 12: *The costs of poorly-chosen timeouts.* For timeouts which are too long, latency increases dramatically. For timeouts which are too short, maintenance messages get lost and the network partitions.

5.3.2 Message timeouts

In the previous section we argued that proactive neighbor recovery can inhibit a DHT's ability to handle churn, motivating the periodic algorithms in Section 4. In this section we give evidence to support the claim of Section 4.2—that accurate message timeout values are also important for handling churn. By using recursive routing, a Bamboo node is able to communicate only with its immediate neighbors, nodes for which it maintains accurate timeouts. This section thus also shows the importance of recursive routing to Bamboo's robustness under churn. As we discussed in Section 4.2, we do not mean to imply that iterative routing cannot work under churn, rather that accurate timeout computation will be essential in iterative routing as well, and that it is less clear how to do it in iterative systems.

To investigate the cost of inaccurate timeouts, we perturbed the timeout values computed by Bamboo's networking layer as follows. As before, the timeouts for each

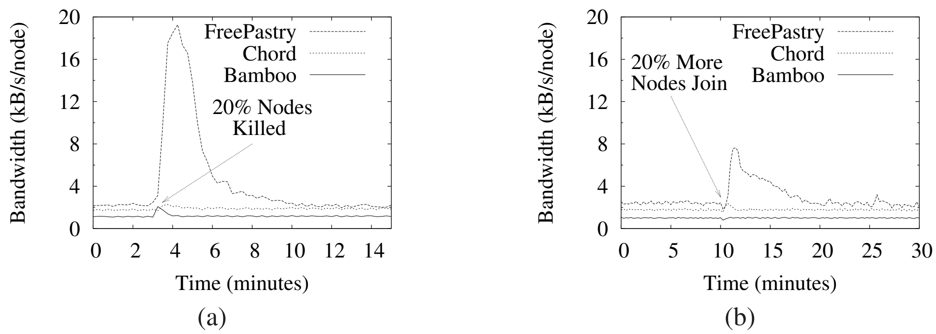


Figure 11: *Proactive vs. periodic recovery.* Pastry performs proactive recovery, which tries to correct inconsistencies as they are discovered, whereas Chord and Bamboo perform periodic recovery. Proactive recovery can lead to congestion collapse if recovery operations overwhelm the available bandwidth.

message are computed in a manner similar to TCP, as a function of the exponentially weighted mean round-trip time and variance thereof. However, in this experiment we also apply a perturbation factor to these timeouts, multiplying each computed timeout by a factor of f. We then apply a 20 minute period of churn with 12-minute median session times and observe the resulting consistency and 95th percentile latency.

Figure 12 shows the results of this experiment. It is clear that a small error to either side of the correct value is sufficient to impair the DHT's ability to handle churn. For timeouts which are too long, latency increases dramatically. For timeouts which are too short, maintenance messages get lost and the network partitions.[9] We conclude from Figure 12 that an absolute timeout error of $10\times$ is sufficient to inhibit the DHTs ability to handle churn.

5.4 Sensitivity to maintenance periods

Our final churn experiment examines the sensitivity of Bamboo's performance under churn to the frequency of its maintenance periods. By default, Bamboo pings each neighbor every 20 seconds, pushes its leaf set to a random neighbor every 4 seconds, and performs local and global turning once every 10 and 20 seconds, respectively. In Figure 13, we show the 5th, 50th, and 95th percentile latencies for lookups under various median session times using the default periods under the title "1x". Also shown is Bamboo's performance when these periods are multiplied by a factor of 2, 4, or 8. When periods are backed off by a factor of 8, median lookup latencies suffer by around a factor of 2.4, but 95th percentile latencies suffer by a factor of 7.3. Not shown are the percentage of lookups that are both completed and consistent, which drops from

[9]If we could separate routing messages from maintenance ones, it is possible we would only see latency growth for too-short timeouts as well. Unfortunately, making this distinction in the Bamboo networking layer is non-trivial; we hope to investigate this claim in the future.

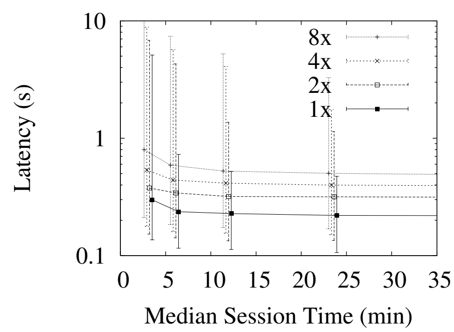


Figure 13: *Sensitivity to maintenance periods.* "1x" shows the data from Figure 10.d. "nx" shows the results with all maintenance periods lengthened by a factor of n.

99% at the highest churn rate shown to 93% at the same churn rate with 8 times less frequent maintenance.

5.5 Static network performance

Finally, we examine the performance of Bamboo on a network without churn. Our goal is to show that the Bamboo routing table maintenance algorithms, designed for high-churn networks, also perform well under low churn. Our goal is the published performance of Pastry, a DHT with the same geometry as Bamboo.

Our experimental setup is as follows. We used an event-driven simulator to run Bamboo nodes over a GT-ITM topology [25]. We used the *ts16384-6* and *ts16384-8* topologies used by Gummadi et al. [10] that were generated with widely different parameters, allowing us to gauge the sensitivity of our algorithms to the underlying network topology. This simulator calculates network delays according to bandwidth and latency values assigned to paths but does not model queuing effects. Since our other experiments have shown that Bamboo demonstrates

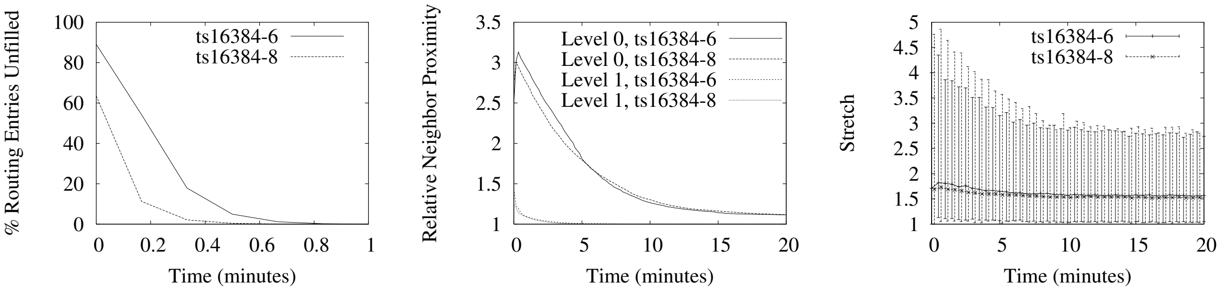


Figure 14: *Bamboo performance in a static network.* From left to right, shown are the percentage of fillable routing table entries that are unfilled, the mean relative proximity of those entries compared to optimal, and the resulting stretch observed for routes through the DHT. Note that the leftmost graph has a different time scale.

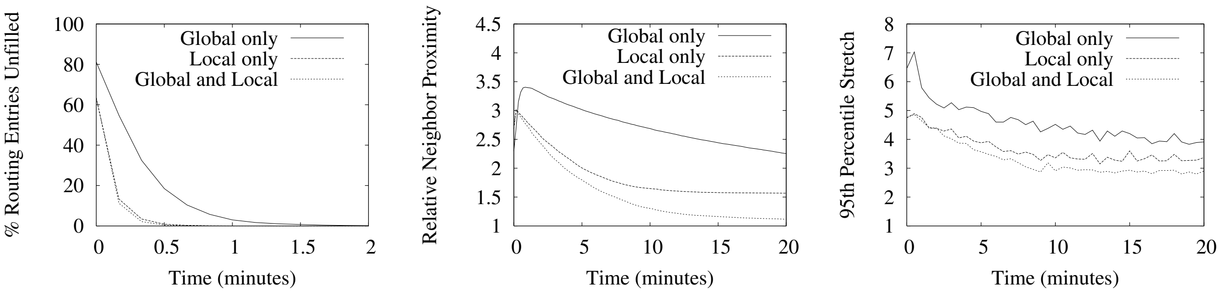


Figure 15: *Effects of global vs. local tuning.* From left to right, shown are the percentage of fillable routing table entries that are unfilled, the mean relative proximity of level-0 entries compared to optimal, and the resulting 95th percentile stretch observed for routes through the DHT. Note that the leftmost graph has a different time scale.

only modest bandwidth usage, we do not believe this limitation should affect the accuracy of our results. To perform this experiment, we simultaneously start 1000 nodes all through the same gateway and observe the routing tables over time. Like Pastry nodes, Bamboo nodes which join through nearby gateways into a network of nodes with already-optimized routing tables will quickly optimize their own routing tables. Since this test uses only one gateway for all nodes, and starts them all at the same time, it is thus a pessimistic assessment of the expected time for a Bamboo node to optimize its routing tables in a static network.

Figure 14 shows the results. On the left, we show the percentage of fillable routing table entries across all nodes which are unfilled as a function of time. When this value goes to zero, Bamboo can route to all destinations using $O(\log N)$ hops in an N-node network. In the center, we show the mean relative proximity of the routing table entries over time. A value of x indicates that the average routing table neighbor is x times further away than the closest available node for that entry. Under certain network assumptions [18] relative proximity of neighbors allows Bamboo to route in time proportional to the underlying network distance. On the right, we confirm this assertion by plotting the aggregate routing stretch. A stretch of x means that a route that started on node A and ended on node B took a path through the physical network that was x times longer than the direct path between A and B. The values shown are the 5th, 50th, and 95th percentiles. The mean stretch after 10 minutes for both topologies is 1.79; for a similar topology Pastry has been shown to have a mean stretch of 1.59, an 11% improvement over Bamboo [5]. Developing new routing table maintenance algorithms to close this performance gap is an important aspect of future work, but in the meantime we view this slight inefficiency as a small cost to pay for resilience to churn.

Our final experiment for static networks explores the relative benefits of the global versus local routing table tuning algorithms. Figure 15 shows the same metrics as Figure 14, but comparing Bamboo's performance using only the global tuning algorithm to its performance using only the local tuning algorithm, versus its performance using both algorithms. The latter configuration is the same as shown in Figure 14. This experiment was run on both topologies with similar results, but to conserve space only the results for *ts16384-8* are shown. We note from the center graph that while local tuning quickly moves the routing tables in the system towards optimality, it cannot find many close neighbors. This symptoms of this deficiency are illustrated in right-hand graph, where the 95th percentile latency is shown suffering. These effects are an illustration of the problem highlighted by Figure 9; there is simply no path from the existing neighbors a node has and the neighbors it needs to optimize its routing table. On the other hand, since global tuning can find any node in the network, but does so only one node at a time, we see that it improves routing tables more slowly; although it is not shown in the graph, so long as the leaf sets are consistent, global tuning will always improve a routing table to optimality, but it make take a long time to do so. The combination of the algorithms creates good routing tables quickly, and will eventually converge them to optimality.

6 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. [10] 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. [14] 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 proactive failure recovery can exacerbate such conditions.

Mahajan et al. [15] present a simulation-based analysis of Pastry. They do not consider the possibility that a Pastry network could become inconsistent (as [14] does for Chord), but instead look at the probability of message loss for various rates of maintenance traffic. Furthermore, they show an algorithm for automatically tuning that rate for a given maximum loss rate; while we have not investigated it, we believe this result could be usefully applied to Bamboo. Unlike our work, they do not consider hop-by-hop retransmissions during lookup. While end-to-end retransmissions are clearly necessary for correctness, choosing a good timeout for an arbitrary lookup is hard. Moreover, we have shown that intermediate nodes in a route are aptly suited for computing retransmission times, and that performance suffers significantly if these times are perturbed. Finally, the simulations of [15] do not consider such network issues as queuing, packet loss, etc.; while this allows them to simulate far larger networks than we have studied here, it does not reveal, for example, the drawbacks of proactive recovery shown by our emulations of smaller networks. 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 the three DHTs studied here. For example, Kademlia [16] 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" [1, 26]. 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.

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, a DHT designed to handle churn, and shown that it is robust under median session times as short as 1.5 minutes, the fastest rate our testing framework can sustain.

Bamboo's resilience to churn is the combined result of several factors. First, the static resilience of its hybrid geometry allows the network to continue functioning after a failure until recovery can occur. This static resilience is a necessary first step to handling failures in general and churn in particular. Second, recovery is performed in a periodic—rather than proactive—manner: the recovery process itself does not further load the network, avoiding a possible positive feedback loop in which congestion is interpreted as further node failure. Third, Bamboo's use of recursive routing allows for active probing of neighbor links, which in turn allows for the calculation of effective response timeouts for requests send over those links. Without such accurate timeout information, a probable node failure is difficult to distinguish from network or processor congestion, and the ability of the DHT to adapt is inhibited. Moreover, in networks with heavy background traffic, we expect this resilience to congestion to be important in handling even lighter levels of churn than those emphasized here. We plan to study such scenarios in our future work.

While we have presented these latter two principles in the context of Bamboo, we believe they have more general applicability to all DHTs with high static resilience. Periodic recovery is a general technique already employed by Chord; Bamboo is simply the first DHT to present an effective set of periodic recovery algorithms that maintain proximity in neighbor selection. Also, the derivation of accurate timeout values need not be based on active probing; it is possible that the coordinate-based timeout calculation schemes can be made to respond quickly enough to changing network conditions to be an effective substitute. If so, we expect DHTs that use iterative routing to handle churn as well as their recursive counterparts. Since iterative routing is attractive for a variety of other reasons, we view these coordinate algorithms as an important area of future research.

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 only a first step. 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. Nonetheless, 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.

References

[1] Gnutella. `http://www.gnutella.com/`, September 2003.

[2] Overnet. `http://www.overnet.com/`, September 2003.

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

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

[5] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft, 2002.

[6] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[7] 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.

[8] R. Cox and F. Dabek. Learning Euclidean coordinates for Internet hosts. `http://pdos.lcs.mit.edu/~rsc/6867.pdf`, Dec. 2002.

[9] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. ACM PODC*, 1987.

[10] 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.

[11] 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.

[12] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram congestion control protocol (DCCP). `http://www.icir.org/kohler/dcp/draft-ietf-dccp-spec-04.txt`, June 2003.

[13] K. Lakshminarayanan, I. Stoica, and S. Shenker. Building a flexible and efficient routing infrastructure: Needs and challenges. Technical Report CSD-03-1254, UC Berkeley, June 2003.

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

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

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

[17] B. Pittel. On spreading a rumor. *SIAM J. Applied Math*, 47, 1987.

[18] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. June 1997.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] E. W. Zegura, K. Calvert, and M. J. Donahoo. A Quantitative Comparison of Graph-based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, Dec. 1997.

[26] 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.

[27] B. Y. Zhao, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.