Bandwidth-efficient management of DHT routing tables

Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek
MIT Computer Science and Artificial Intelligence Laboratory
{jinyang, strib, rtm, kaashoek}@csail.mit.edu

Abstract

Today an application developer using a distributed hash table (DHT) with n nodes must choose a DHT protocol from the spectrum between $O(1)$ lookup protocols [9, 18] and $O(\log n)$ protocols [20–23, 25, 26]. $O(1)$ protocols achieve low latency lookups on small or low-churn networks because lookups take only a few hops, but incur high maintenance traffic on large or high-churn networks. $O(\log n)$ protocols incur less maintenance traffic on large or high-churn networks but require more lookup hops in small networks. Accordion is a new routing protocol that does not force the developer to make this choice: Accordion adjusts itself to provide the best performance across a range of network sizes and churn rates while staying within a bounded bandwidth budget.

The key challenges in the design of Accordion are the algorithms that choose the routing table's size and content. Each Accordion node learns of new neighbors opportunistically, in a way that causes the density of its neighbors to be inversely proportional to their distance in ID space from the node. This distribution allows Accordion to vary the table size along a continuum while still guaranteeing at most $O(\log n)$ lookup hops. The user-specified bandwidth budget controls the rate at which a node learns about new neighbors. Each node limits its routing table size by evicting neighbors that it judges likely to have failed. High churn (*i.e.*, short node lifetimes) leads to a high eviction rate. The equilibrium between the learning and eviction processes determines the table size.

Simulations show that Accordion maintains an efficient lookup latency versus bandwidth tradeoff over a wider range of operating conditions than existing DHTs.

1 Introduction

Distributed hash tables maintain routing tables used when forwarding lookups. A node's routing table consists of a set of "neighbor" entries, each of which contains the IP address and DHT identifier of some other node. A DHT node must maintain its routing table, both populating it initially and ensuring that the neighbors it refers to are still alive.

Existing DHTs use routing table maintenance algorithms that work best in particular operating environments. Some maintain small routing tables in order to limit the maintenance communication cost [11, 20–23, 25, 26]. Small tables help the DHT scale to many nodes and limit the maintenance required if the node population increases rapidly. The disadvantage of a small routing table is that lookups may take many time-consuming hops, typically $O(\log n)$ in a system with n nodes.

At the other extreme are DHTs that maintain a complete list of nodes in every node's routing table [9, 18]. A large routing table allows single-hop lookups. However, each node must promptly learn about every node that joins or leaves the system, as otherwise lookups are likely to experience frequent timeout delays due to table entries that point to dead nodes. Such timeouts are expensive in terms of increased end-to-end lookup latency [2, 16, 22]. The maintenance traffic needed to avoid timeouts in such a protocol may be large if there are many unstable nodes or the network size is large.

An application developer wishing to use a DHT must choose a protocol between these end points. An $O(1)$ protocol might work well early in the deployment of an application, when the number of nodes is small, but could generate too much maintenance traffic as the application becomes popular or if churn increases. Starting with an $O(\log n)$ protocol would result in unnecessarily low performance on small networks or if churn turns out to be low. While the developer can manually tune a $O(\log n)$ protocol to increase the size of its routing table, such tuning is difficult and workload-dependent [16].

This paper describes a new DHT design, called Accordion, that automatically tunes parameters such as routing table size in order to achieve the best performance. Accordion has a single parameter, a network bandwidth budget, that allows control over the consumption of the resource that is most constrained for typical users. Given the budget, Accordion adapts its behavior across a wide range of network sizes and churn rates to provide low-latency lookups.

This research was conducted as part of the IRIS project (http://project-iris.net/), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660. Jinyang Li is also supported by a Microsoft Research Fellowship.

The problems that Accordion must solve are how to arrive at the best routing table size in light of the budget and the stability of the node population, how to choose the most effective neighbors to place in the routing table, and how to divide the maintenance budget between acquiring new neighbors and checking the liveness of existing neighbors.

Accordion solves these problems in a unique way. Unlike other protocols, it is not based on a particular data structure such as a hypercube or de Bruijn graph that constrains the number and choice of neighbors. Instead, each node learns of new neighbors as a side-effect of ordinary lookups, but selects them so that the density of its neighbors is inversely proportional to their distance in ID space from the node. This distribution allows Accordion to vary the table size along a continuum while still providing the same worst-case guarantees as traditional $O(\log n)$ protocols. A node's bandwidth budget determines the rate at which a node learns. Each node limits its routing table size by evicting neighbors that it judges likely to have failed: those which have been up for only a short time or have not been heard from for a long time. Therefore, high churn leads to a high eviction rate. The equilibrium between the learning and eviction processes determines the table size.

Performance simulations show that Accordion keeps its maintenance traffic within the budget over a wide range of operating conditions. When bandwidth is plentiful, Accordion provides lookup latencies and maintenance overhead similar to that of OneHop [9]. When bandwidth is scarce, Accordion has lower lookup latency and less maintenance overhead than Chord [5, 25], even when Chord incorporates proximity and has been tuned for the specific workload [16].

The next two sections outline Accordion's design approach and analyze the relationship between maintenance traffic and table size. Section 4 describes the details of the Accordion protocol. Section 5 compares Accordion's performance with that of other DHTs. Section 6 presents related work, and Section 7 concludes.

2 Design Challenges

A DHT's routing table maintenance traffic must fit within the nodes' access link capacities. Most existing designs do not live within this physical constraint. Instead, the amount of maintenance traffic they consume is determined as a side effect of the total number of nodes and the rate of churn. While some protocols (*e.g.*, Bamboo [22] and MSPastry [2]) have mechanisms for limiting maintenance traffic during periods of high churn or congestion, one of the goals of Accordion is to keep this traffic within a budget determined by link capacity or user preference.

Once a DHT node has a maintenance budget, it must decide how to use the budget to minimize lookup latency. This latency depends largely on two factors: the average number of hops per lookup and the average number of timeouts incurred during a lookup. A node can choose to spend its bandwidth budget to aggressively maintain the freshness of a smaller routing table (thus minimizing timeouts), or to look for new nodes to enlarge the table (thus minimizing lookup hops but perhaps risking timeouts). Nodes may also use the budget to issue lookup messages along multiple paths in parallel, to mask the effect of timeouts occurring on any one path. Ultimately, the bandwidth budget's main effect is on the size and contents of the routing table.

Rather than explicitly calculating the best table size based on a given budget and an observed churn rate, Accordion's table size is the result of an equilibrium between two processes: state acquisition and state eviction. The state acquisition process learns about new neighbors; the bigger the budget is, the faster a node can learn, resulting in a bigger table size. The state eviction process deletes routing table entries that are likely to cause lookup timeouts; the higher the churn, the faster a node evicts state. The next section investigates and analyzes budgeted routing table maintenance issues in more depth.

3 Table Maintenance Analysis

In order to design a routing table maintenance process that makes the most effective use of the bandwidth budget, we have to address three technical questions:

1. How do nodes choose neighbors for inclusion in the routing table in order to guarantee at most $O(\log n)$ lookups across a wide range of table sizes?
2. How do nodes choose between active exploration and opportunistic learning (perhaps using parallel lookups) to learn about new neighbors in the most efficient way?
3. How do nodes evict neighbors from the routing table with the most efficient combination of active probing and uptime prediction?

3.1 Routing State Distribution

Each node in a DHT has a unique identifier, typically 128 or 160 random bits generated by a secure hash function. Structured DHT protocols use these identifiers to assign responsibility for portions of the identifier space. A node keeps a routing table that points to other nodes in the network, and forwards a query to a neighbor based on the neighbor's identifier and the lookup key. In this manner, the query gets "closer" to the node responsible for the key in each successive hop.

A DHT's *routing structure* determines from which regions of identifier space a node chooses its neighbors. The ideal routing structure is both flexible and scalable. With a flexible routing structure, a node is able to expand and contract the size of the routing table along a continuum in response to churn and bandwidth budget. With a scalable routing structure, even a very small routing table can lead to efficient lookups in a few hops. However, as currently defined, most DHT routing structures are scalable but not flexible and constrain which routing table sizes are possible. For example, a Tapestry node with a 160-bit identifier of base b maintains a routing table with $\frac{160}{\log_2 b}$ levels, each of which contain $b - 1$ entries. In practice, few of these levels are filled, and the expected number of neighbors per node in a network of n DHT nodes is $(b-1)\log_b n$. The parameter base (b) controls the table size, but it can only take values that are powers of 2, making it difficult to adjust the table size smoothly.

Existing routing structures are rigid in the sense that they *require* neighbors from certain regions of ID space to be present in the routing table. We can relax the table structure by specifying only the distribution of ID space distances between a node and its neighbors. Viewing routing structure as a probabilistic distribution gives a node the flexibility to use a routing table of any size. We model the distribution after proposed scalable routing structures. The ID space is organized as a ring as in Chord [25] and we define the ID distance to be the clockwise distance between two nodes on the ring.

Accordion uses a $\frac{1}{x}$ distribution to choose its neighbors: the probability of a node selecting a neighbor with distance x from itself in the identifier space from itself is proportional to $\frac{1}{x}$. This distribution causes a node to prefer neighbors that are closer to itself in ID space, ensuring that as a lookup gets closer to the target key there is always likely to be a helpful routing table entry. This $\frac{1}{x}$ distribution is the same as the "small-world" model proposed by Kleinberg [13], previously used by DHTs such as Symphony [19] and Mercury [1]. The $\frac{1}{x}$ distribution is also scalable and results in $O(\frac{\log n \log\log n}{\log s})$ lookup hops if each node has a table size of s; this result follows from an extension of Kleinberg's analysis [13].

3.2 Routing State Acquisition

A straightforward approach to learning new neighbors is to search actively for nodes with the $\frac{1}{x}$ distribution. A more bandwidth-efficient approach, however, is to learn about new neighbors, and the liveness of existing neighbors, as a side-effect of ordinary lookup traffic.

Learning through lookups does not necessarily yield useful information about existing neighbors or about new neighbors with the desired distribution in ID space. For example, if the DHT used *iterative routing* [25] during lookups, the original querying node would talk directly to each hop of the lookup. Assuming the keys being looked up are uniformly distributed, the querying node would communicate with nodes in a uniform distribution rather than a $\frac{1}{x}$ distribution.

With *recursive routing*, on the other hand, intermediate hops of a lookup forward the lookup message directly to the next hop. This means that nodes communicate only with existing neighbors from their routing tables during lookups. If each hop of a recursive lookup is acknowledged, then a node can check the liveness of a neighbor with each lookup it forwards, and the neighbor can piggyback information about its own neighbors in the acknowledgment.

If lookup keys are uniformly distributed and the nodes already have routing tables following a small-world distribution, then each lookup will involve one hop at exponentially smaller intervals in identifier space. Therefore, a node forwards lookups to next-hop nodes that fit its small-world distribution. A node can then learn about entries immediately following the next-hop nodes in identifier space, ensuring that the new neighbors learned also follow this distribution.

In practice lookup keys are not necessarily uniformly distributed, and thus Accordion devotes a small amount of its bandwidth budget to actively exploring for new neighbors according to the small-world distribution.

A DHT can learn even more from lookups if it performs parallel lookups, by sending out multiple copies of each lookup down different lookup paths. This increases the opportunity to learn new information, while at the same time decreasing lookup latency by circumventing potential timeouts. Analysis of DHT design techniques show that learning extra information from parallel lookups is more efficient at lowering lookup latencies than checking existing neighbor liveness or active exploration [16]. Accordion adjusts the degree of lookup parallelism based on the current lookup load to stay within the specified bandwidth budget.

3.3 Routing State Freshness

A DHT node must strike a balance between the freshness and the size of its routing table. While parallel lookups can help mask timeouts caused by stale entries, nodes still need to judge the freshness of entries to decide when to evict nodes, in order to limit the number of expected lookup timeouts.

Timeouts are expensive as nodes need to wait multiple round trip times to declare the lookup message failed before re-issuing it to a different neighbor [2, 22]. In order to avoid timeouts, most existing DHTs [2, 5, 20, 26] contact each neighbor periodically to determine the routing entry's liveness. In other words, a node can control its routing state freshness by evicting neighbors from its routing table that it has not successfully contacted for some interval. If the bandwidth budget were infinite, the node could ping each neighbor often to maintain fresh tables of arbitrarily large

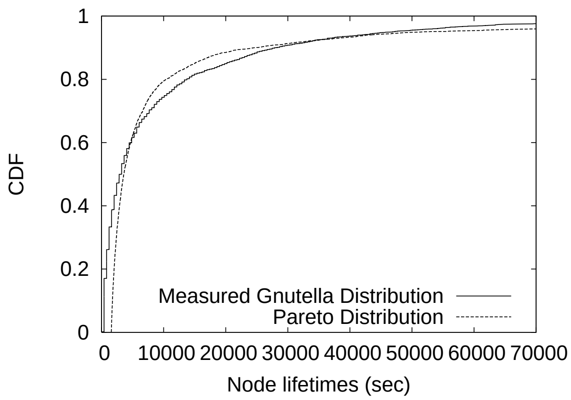

Figure 1: Cumulative distribution of measured Gnutella node uptime [24] compared with a Pareto distribution using $\alpha = 0.83$ and $\beta = 1560$ sec.

size. However, with a finite bandwidth, a DHT node must somehow make a tradeoff between the freshness and the size of its routing table. This section describes how to predict the freshness of routing table entries so that entries can be evicted efficiently.

3.3.1 Characterizing Freshness

The freshness of a routing table entry can be characterized with p, the probability of a neighbor being alive. The eviction process deletes a neighbor from the table if the estimated probability of it being alive is below some threshold p_{thresh}. Therefore, we are interested in finding a value for p_{thresh} such that the total number of lookup hops including timeout retries are minimized. If node lifetimes follow a memoryless exponential distribution, p is determined only by Δt_{since}, where Δt_{since} is the time interval since the neighbor was last known to be alive. However, in real systems, the distribution of node lifetimes is often heavy-tailed: nodes that have been alive for a long time are more likely to stay alive for an even longer time. In a heavy-tailed Pareto distribution, for example, the probability of a node dying before time t is

$$\Pr(\textit{lifetime} < t) = 1 - \left(\frac{\beta}{t}\right)^{\alpha}$$

where α and β are the shape and scale parameters of the distribution, respectively. Saroiu et al. measure such a distribution in a study of the Gnutella network [24]; in Figure 1 we compare their measured Gnutella lifetime distribution with a synthetic heavy-tailed Pareto distribution (using $\alpha = .83$ and $\beta = 1560$ sec). In a heavy-tailed distribution, p is determined by both the time when the node joined the network, Δt_{alive}, and Δt_{since}. We will present our estimation of p_{thresh} assuming a Pareto distribution for node lifetimes.

Let Δt_{alive} be the time for which the neighbor has been a member of the DHT, measured at the time it was last heard, Δt_{since} seconds ago. The conditional probability of a neighbor being alive, given that it had already been alive for Δt_{alive} seconds, is

$$p = \Pr(\textit{lifetime} > (\Delta t_{alive} + \Delta t_{since}) \mid \textit{lifetime} > \Delta t_{alive})$$

$$= \frac{(\frac{\beta}{\Delta t_{alive}+\Delta t_{since}})^{\alpha}}{(\frac{\beta}{\Delta t_{alive}})^{\alpha}} = \left(\frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}}\right)^{\alpha} \qquad (1)$$

Therefore, $\Delta t_{since} = \Delta t_{alive}(p^{-\frac{1}{\alpha}} - 1)$. Since Δt_{alive} follows a Pareto distribution, the median lifetime is $2^{\frac{1}{\alpha}}\beta$. Therefore, within $\Delta t_{since} = 2^{\frac{1}{\alpha}}\beta(p_{thresh}^{-\frac{1}{\alpha}} - 1)$ seconds, half of the routing table should be evicted with the eviction threshold set at p_{thresh}. If s_{tot} is the total routing table size, the eviction rate is approximately $\frac{s_{tot}}{2\Delta t_{since}}$.

Since nodes aim to keep their maintenance traffic below a certain bandwidth budget, they can only refresh or learn about new neighbors at some finite rate determined the budget. For example, if a node's bandwidth budget is 20 bytes per second, and learning liveness information for a single neighbor costs 4 bytes (*e.g.*, the neighbor's IP address), then at most a node could refresh or learn routing table entries for 5 nodes per second.

Suppose that a node has a bandwidth budget such that it can afford to refresh/learn about B nodes per second. The routing table size s_{tot} at the equilibrium between eviction and learning is:

$$\frac{s_{tot}}{2\Delta t_{since}} = B$$

$$\Rightarrow s_{tot} = 2B\Delta t_{since} = 2B(2^{\frac{1}{\alpha}})\beta(p_{thresh}^{-\frac{1}{\alpha}} - 1) \qquad (2)$$

However, some fraction of the table points to dead neighbors and therefore does not contribute to lowering lookup hops. The *effective* routing table size, then, is $s = s_{tot} \cdot p_{thresh}$.

3.3.2 Choosing the Best Eviction Threshold

Our goal is to choose a p_{thresh} that will minimize the expected number of hops for each lookup. We know from Section 3.1 that the average number of hops per lookup in a static network is $O(\frac{\log n \log\log n}{\log s})$; under churn, however, each hop successfully taken has an extra cost associated with it, due to the possibility of forwarding lookups to dead neighbors. When each neighbor is alive with probability at least p_{thresh}, the upper bound on the expected number of trials per successful hop taken is $\frac{1}{p_{thresh}}$ (for now, we assume no parallelism). Thus, we can approximate the expected number of actual hops per lookup, h, by multiplying the number

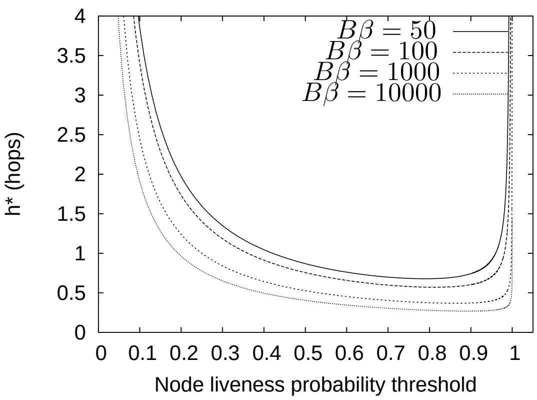

Figure 2: The function h^* (Equation 4) with respect to p_{thresh}, for different values of $B\beta$ and fixed $\alpha = 1$. h^* goes to infinity as p_{thresh} approaches 1.

of effective lookup hops with the expected number of trials needed per effective hop:

$$h \propto \frac{\log n \log\log n}{\log s} \cdot \frac{1}{p_{thresh}}$$

We then substitute the effective table size s with $s_{tot} \cdot p_{thresh}$, using Equation 2:

$$h \propto \frac{\log n \log\log n}{\log(2B\beta(2^{\frac{1}{\alpha}})(p_{thresh}^{-\frac{1}{\alpha}} - 1) \cdot p_{thresh})} \cdot \frac{1}{p_{thresh}} \quad (3)$$

The numerator of Equation 3 is constant with respect to p_{thresh}, and therefore can be ignored for the purposes of minimization. It usually takes on the order of a few round-trip times to detect lookup timeout and this multiplicative timeout penalty can also be ignored. Our task now is to choose a p_{thresh} that will minimize:

$$h^* = \frac{1}{\log(2B\beta(2^{\frac{1}{\alpha}})(p_{thresh}^{-\frac{1}{\alpha}} - 1)p_{thresh}) \cdot p_{thresh}} \quad (4)$$

The minimizing p_{thresh} depends on the constants $(B\beta) \cdot (2^{\frac{1}{\alpha}})$ and α. If p_{thresh} varied widely given different values of $B\beta$ and α, nodes would constantly need to reassess their estimates of p_{thresh} using rough estimates of the current churn rate and the bandwidth budget. Fortunately, this is not the case.

Figure 2 plots h^* with respect to p_{thresh}, for various values of $B\beta$ and a fixed α. We consider only values of $B\beta$ large enough to allow nodes to maintain a reasonable number of neighbors under the given churn rate. For example, if nodes have mean lifetimes of 10 seconds ($\beta = 5$ sec, $\alpha = 1$), but can afford to refresh/learn one neighbor per second, no value of p_{thresh} will allow s to be greater than 2.

Figure 2 shows that as p_{thresh} increases the expected lookup hops decreases due to fewer timeouts; however, as p_{thresh} becomes even larger and approaches 1, the number of hops actually increases due to a limited table size. The p_{thresh} that minimizes lookup hops lies somewhere between .7 and .9 for all curves. Figure 2 also shows that as $B\beta$ increases, the p_{thresh} that minimizes h^* increases as well, but only slightly. In fact, for any reasonable value of $B\beta$, h^* varies so little around its true minimum that we can approximate the optimal p_{thresh} for any value of $B\beta$ to be .9. A similar analysis shows the same results for reasonable α values. For the remainder of this paper, we assume $p_{thresh} = .9$, because even though this may not be precisely optimal, it will produce an expected number of hops that is nearly minimal in most deployment scenarios.

The above analysis for p_{thresh} assumes no lookup parallelism. If lookups are sent down multiple paths concurrently, nodes can use a much smaller value for p_{thresh} because the probability will be small that *all* of the next-hop messages will timeout. Using a smaller value for p_{thresh} leads to a larger effective routing table size, reducing the average lookup hop count. Nodes can choose a p_{thresh} value such that the probability that at least one next-hop message will not fail is at least .9.

3.3.3 Calculating Entry Freshness

Nodes can use Equation 1 to calculate p, the probability of a neighbor being alive, and then evict entries with $p < p_{thresh}$. Calculating p requires estimates of three values: Δt_{alive} and Δt_{since} for the given neighbor, along with the shape parameter α of the Pareto distribution. Interestingly, p does not depend on the scale parameter β, which determines the median node lifetime in the system. This is counterintuitive; we expect that smaller median node lifetimes (*i.e.*, faster churn rates) will decrease p and increase the eviction rate. This median lifetime information, however, is implicitly present in the observed values for Δt_{alive} and Δt_{since}, so β is not explicitly required to calculate p.

Equation 1, as stated, still requires some estimate for α, which may be difficult to observe and calculate. To simplify this task, we define an indicator variable i for each routing table entry as follows:

$$i = \frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}} \quad (5)$$

Since $p = i^{\alpha}$, a monotonically increasing function of i, there exists some i_{thresh} such that any routing table entry with $i < i_{thresh}$ will also have a $p < p_{thresh}$. Thus, if nodes can estimate the value of i_{thresh} corresponding to p_{thresh}, no estimate of α is necessary. All entries with i less than i_{thresh} will be evicted. Section 4.6 describes how Accordion estimates an appropriate i_{thresh} for the observed churn, and how nodes learn Δt_{alive} and Δt_{since} for each entry.

4 The Accordion Protocol

Accordion uses consistent hashing [12] in a circular identifier space to assign keys to nodes. Accordion borrows Chord's protocols for maintaining a linked list from each node to the ones immediately following in ID space (Chord's successor lists and join protocol). An Accordion node's routing table consists of a set of neighbor entries, each containing a neighboring node's IP address and ID.

An Accordion lookup for a key finds the key's successor: the node whose ID most closely follows the key in ID space. When node n_0 starts a query for key k, n_0 looks in its routing table for the neighbor n_1 whose ID most closely precedes k, and sends a query packet to n_1. That node follows the same rule: it forwards the query to the neighbor n_2 that most closely precedes k. When the query reaches node n_i and k lies between n_i and the n_i's successor, the query has finished; n_i sends a reply directly back to n_0 with the identity of its successor (the node responsible for k).

4.1 Bandwidth Budget

Accordion's strategy for using the bandwidth budget is to use as much bandwidth as possible on lookups by exploring multiple paths in parallel [16]. When some bandwidth is left over (perhaps due to bursty lookup traffic), Accordion uses the rest to explore; that is, to find new routing entries according to a small-world distribution.

This approach works well because parallel lookups serve two functions. Parallelism reduces the impact of timeouts on lookup latency because one copy of the lookup may proceed while other copies wait in timeout. Parallel lookups also allow nodes to learn about new nodes and about the liveness of existing neighbors, and as such it is better to learn as a side-effect of lookups than from explicit probing. Section 4.3 explains how Accordion controls the degree of lookup parallelism to try to fill the whole budget.

Accordion must also keep track of how much of the budget is left over and available for exploration. To control the budget, each node maintains an integer variable, b_{avail}, which keeps track of the number of bytes available to the node for exploration traffic, based on recent activity. Each time the node sends a packet or receives the corresponding acknowledgment (for any type of traffic), it decrements b_{avail} by the size of the packet. It does not decrement b_{avail} for unsolicited incoming traffic, or for the corresponding outgoing acknowledgments. In other words, each packet only counts towards the bandwidth budget at one end. Periodically, the node increments b_{avail} at the rate of the bandwidth budget.

The user gives the bandwidth budget in two parts: the average desired rate of traffic in bytes per second (r_{avg}), and the maximum burst size in bytes (b_{burst}). Every t_{inc} seconds, the node increments b_{avail} by $r_{avg} \cdot t_{inc}$ (where t_{inc} is the size of one exploration packet divided by r_{avg}). Whenever b_{avail} is positive, the node sends one exploration packet, according to the algorithm we present in Section 4.4. Nodes decrement b_{avail} down to a minimum of $-b_{burst}$. While $b_{avail} = -b_{burst}$, nodes immediately stop sending all low priority traffic (such as redundant lookup traffic and exploration traffic). Thus, nodes send no exploration traffic unless the average traffic over the last b_{burst}/r_{avg} seconds has been less than r_{avg}.

The bandwidth budget controls the maintenance traffic sent by an Accordion node, but does not give the node direct control over all incoming and outgoing traffic. For example, a node must acknowledge all traffic sent to it from its predecessor regardless of the value of b_{avail}; otherwise, its predecessor may think it has failed and the correctness of lookups would be compromised. The imbalance between a node's specified budget and its actual incoming and outgoing traffic is of special concern in scenarios where nodes have heterogeneous budgets in the system. To help nodes with low budgets avoid excessive incoming traffic from nodes with high budgets, an Accordion node biases lookup and table exploration traffic toward neighbors with higher budgets. Section 4.5 describes the details of this bias.

4.2 Learning from Lookups

When an Accordion node forwards a lookup (see Figure 4.2), the immediate next-hop node returns an acknowledgment that includes a set of neighbors from its routing table; this acknowledgment allows nodes to learn from lookups. The acknowledgment also serves to indicate that the next-hop is alive.

If n_1 forwards a lookup for key k to n_2, n_2 returns a set of neighbors in the ID range between n_2 and k. Acquiring new entries this way allow nodes to preferentially learn about ID spaces close-by to itself, the key characteristic of a small-world distribution. Additionally, the fact that n_1 forwarded the lookup to n_2 indicates that n_1 does not know of any nodes in the ID gap between n_2 and k, and n_2 is well-situated to fill this gap.

4.3 Parallel Lookups

An Accordion node increases the parallelism of lookups it initiates and forwards until the point where the lookup traffic nearly fills the bandwidth budget. An Accordion node must adapt the level of parallelism as the underlying lookup rate changes, it must avoid forwarding the same lookup twice, and it must choose the most effective set of nodes to which to forward copies of each lookup.

A key challenge in Accordion's parallel lookup design is caused by its use of recursive routing. Previous DHTs with parallel lookups use iterative routing: the originating node sends lookup messages to each hop of the lookup in

```
procedure NextHop(lookup_request q)
  if this node owns q.key then {
      reply to lookup source directly
      return (NULL)
  }
  // use bias to pick best predecessor (Section 4.5)
  nexthop ← routetable.BestPred(q.key)
  // forward query to next hop
  // and wait for ACK and learning info
  nextreply ← nexthop.NextHop(q)
  put nodes of nextreply in routetable
  // find some nodes between this node
  // and the key, and return them
  return (GetNodes(q.lasthop, q.key))

procedure GetNodes(src, end)
  s ← neighbors between me and end
  // m is some constant (e.g., 5)
  if s.SIZE() < m then v ← s
  else v ← m nodes in s nearest to src w.r.t. latency
  return (v)
```

Figure 3: Learning from lookups in Accordion.

turn [15,20]. Iterative lookups allow the originating node to explicitly control the amount of parallelism and the order in which paths are explored, since the originating node issues all messages related to the lookup. However, Accordion uses recursive routing to learn nodes with a small-world distribution, and nodes forward lookups directly to the next hop. To control recursive parallel lookups, each Accordion node independently adjusts its lookup parallelism to stay within the bandwidth budget.

If an Accordion node knew the near-term future rate at which it was about to receive lookups to be forwarded, it could divide the bandwidth budget by that rate to determine the level of parallelism. Since it cannot predict the future, Accordion uses an adaptive algorithm to set the level of parallelism based on the past lookup rate. Each node maintains a w_p "parallelism window" variable that determines the number of copies it forwards of each received or initiated lookup. A node updates w_p every t_p seconds, where $t_p = b_{burst}/r_{avg}$, which allows enough time for the bandwidth budget to recover from potential bursts of lookup traffic. During each interval of t_p seconds, a node keeps track of how many unique lookup packets it has originated or forwarded, and how many exploration packets it has sent. If more exploration packets have been sent than the number of lookups that have passed through this node, w_p increases by 1. Otherwise, w_p decreases by half. This additive increase/multiplicative decrease (AIMD) style of control ensures a prompt response to w_p overestimation or sudden changes in the lookup load. Additionally, nodes do not increase w_p above some maximum value, as determined by the maximum burst size, b_{burst}. A node forwards the w_p copies of a lookup to the w_p neighbors whose IDs most closely precede the desired key in ID space.

When a node originates a query, it marks one of the parallel copies with a "primary" flag which gives that copy high priority. Intermediate nodes are free to drop non-primary copies of a query if they do not have sufficient bandwidth to forward the query, or if they have already seen a copy of the query in the recent past. If a node receives a primary query, it marks one forwarded copy as primary, maintaining the invariant that there is always one primary copy of a query. Primary lookup packets trace the path a non-parallel lookup would have taken, while non-primary traffic copies act as optional traffic to decrease timeout latency and increase information learned.

4.4 Routing Table Exploration

When lookup traffic is bursty, Accordion might not be able to accurately predict w_p for the next time period. As such, parallel lookups would not consume the entire bandwidth budget during that time period. Accordion uses this leftover bandwidth to explore for new neighbors actively. Because lookup keys are not necessarily distributed uniformly in practice, a node might not be able to learn new entries with the correct distribution through lookups alone; explicit exploration addresses this problem. The main goal of exploration is that it be bandwidth-efficient and result in learning nodes with the small-world distribution described in Section 3.1.

For each neighbor x ID-distance away from a node, the gap between that neighbor and the next successive entry should be proportional to x. A node with identifier a compares the *scaled gaps* between successive neighbors n_i and n_{i+1} to decide the portion of its routing table most in need of exploration. The scaled gap g between neighbors n_i and n_{i+1} is:

$$g = \frac{d(n_i, n_{i+1})}{d(a, n_i)}$$

where $d(x, y)$ computes the clockwise distance in the circular identifier space between identifiers x and y. When an Accordion node sends an exploration query, it sends it to the neighbor with the largest scaled gap between it and the next neighbor. The result is that the node explores in the area of ID space where its routing table is the most sparse with respect to the desired distribution.

An exploration message from node a asks neighbor n_i for m neighbor entries between n_i and n_{i+1}, where m is some small constant (*e.g.*, 5). n_i retrieves these entries from both its successor list and its routing table. n_i uses Vivaldi network coordinates [4] to find the m nodes in this gap with the lowest predicted network delay to a. If n_i returns fewer than m entries, node a will not revisit n_i again until it has explored all other neighbors.

The above process only *approximates* a $\frac{1}{x}$ distribution; it does not guarantee such a distribution in all cases. Such a guarantee would not be flexible enough to allow a full routing table when bandwidth is plentiful and churn is low. Accordion's exploration method results in a $\frac{1}{x}$ distribution when churn is high, but also achieves nearly full routing tables when the bandwidth budget allows.

4.5 Biasing Traffic to High-Budget Nodes

Because nodes have no direct control over their incoming bandwidth, in a network containing nodes with diverse bandwidth budgets we expect that some nodes will be forced over-budget by incoming traffic from nodes with bigger budgets. Accordion addresses this budgetary imbalance by biasing lookup and exploration traffic toward nodes with higher budgets. Though nodes still do not have direct control over their incoming bandwidth, in the absence of malicious nodes this bias serves to distribute traffic in proportion to the bandwidth budgets of nodes.

When an Accordion node learns about a new neighbor, it also learns that neighbor's bandwidth budget. Traditional DHT protocols (*e.g.*, Chord) route lookups greedily to the neighbor most closely preceding the key in ID space, because that neighbor is expected to have the highest density of routing entries near the key. We generalize this idea to consider bandwidth budget. Since the density of routing entries near the desired ID region increases linearly with the node's bandwidth budget but decreases with the node's distance from that region in ID space, neighbors should forward lookup/exploration traffic to the neighbor with the best *combination* of high budget and short distance.

Suppose a node a decides to send an exploration packet to its neighbor n_1 (with budget b_1), to learn about new entries in the gap between n_1 and the following entry n_0 (as discussed in Section 4.4). Let x be the distance in identifier space between n_1 and the following entry n_0. Let n_i ($i = 2, 3...$) be neighbors preceding n_1 in the a's routing table, each with a bandwidth budget of b_i. In Accordion's traffic biasing scheme, a prefers to send the exploration packet to the neighbor n_i ($i = 1, 2...$) with the largest value for the following equation:

$$v_i = \frac{b_i}{d(n_i, n_1) + x}$$

where $x = d(n_1, n_0)$. In the case of making lookup forwarding decisions for some key k, $x = d(n_1, k)$ and n_1 is the entry immediately precedes k in a's routing table. For each lookup and exploration decision, an Accordion node examines a fixed number of candidate neighbors (set to 8 in our implementation) preceding n_1 and also ensures that the lookup progresses at least halfway towards the key if possible.

To account for network proximity, Accordion further weights the v_i values by the estimated network delay to the neighbor based on network coordinates. With this extension, a chooses the neighbor with the largest value for $v'_i = v_i/delay(a, n_i)$. This is similar in spirit to traditional proximity routing schemes [7].

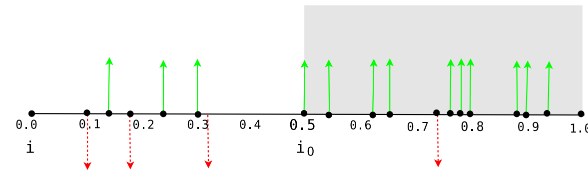

Figure 4: A list of contact entries, sorted by increasing i values. Up arrows indicate events where the neighbor was alive, and down arrows indicate the opposite. A node estimates i_0 to be the minimum i such that there are more than 90% (p_{thresh}) live contacts for $i > i_0$, and then incorporates i_0 into its i_{thresh} estimate.

4.6 Estimating Liveness Probabilities

In order to avoid timeout delays during lookups, an Accordion node must ensure that the neighbors in its routing table are likely to be alive. Accordion does this by estimating each neighbor's probability of being alive, and evicting neighbors judged likely to be dead. For any reasonable node lifetime distribution, the probability that a node is alive decreases as the amount of time since the node was last heard from increases. Accordion attempts to calculate this probability explicitly.

Section 3.3 showed that for a Pareto node lifetime distribution, nodes should evict all entries whose probability of being alive is less than some threshold p_{thresh} so the probability of successfully forwarding a lookup is greater than .9 given the current lookup parallelism w_p (*i.e.*, $1-(1-p_{thresh})^{w_p} = 0.9$). The value i from Equation 5 indicates the probability p of a neighbor being alive. The overall goal of Accordion's node eviction policy is to estimate a value for i_{thresh}, such that nodes evict any neighbor with an associated i value below i_{thresh}. See Section 3.3 for the definitions of i and i_{thresh}.

A node estimates i_{thresh} as follows. Each time it contacts a neighbor, it records whether the neighbor is alive or dead and the neighbor's current indicator value i. Periodically, a node reassesses its estimation of i_{thresh} using this list. It first sorts all the entries in the list by increasing i value, and then determines the smallest value i_0 such that the fraction of entries with an "alive" status and an $i > i_0$ is p_{thresh}. The node then incorporates i_0 into its current estimate of i_{thresh}, using an exponentially-weighted moving average. Figure 4 shows the correct i_0 value for a given sorted list of entries.

To calculate i for each neighbor using Equation 5, nodes must know Δt_{alive} (the time between when the neighbor last joined the network and when it was last heard) and Δt_{since} (the time between when it was last heard and now). Each node keeps track of its own Δt_{alive} based on the time of its last join, and includes its own Δt_{alive} in every packet it sends. Nodes learn (Δt_{alive}, Δt_{since}) information associated with neighbors in one of the following three ways:

- When the node hears from a neighbor directly, it records the current local timestamp as t_{last} in the routing entry for that neighbor, and resets an associated Δt_{since} value to 0 and sets Δt_{alive} to the newly-received Δt_{alive} value.
- If a node hears information about a new neighbor indirectly from another node, it will save the supplied Δt_{since} value in the new routing entry, and set the entry's t_{last} value to the current local timestamp.
- If a node hears information about an existing neighbor, it compares the received Δt_{since} value with its currently recorded value for that neighbor. A smaller received Δt_{since} indicates fresher information about this neighbor, and so the node saves the corresponding (Δt_{alive}, Δt_{since}) pair for the neighbor in its routing table. It also sets t_{last} to the current local timestamp.

Whenever a node needs to calculate a current value for Δt_{since} (either to compare its freshness, to estimate i, or to pass it to a different node), it adds the saved Δt_{since} value and the difference between the current local timestamp and t_{last}.

5 Evaluation

This section demonstrates the important properties of Accordion through simulation. It shows that Accordion matches the performance of existing $\log n$-routing-table DHTs when bandwidth is scarce, and the performance of large-table DHTs when bandwidth is plentiful under different lookup workloads. Accordion achieves low latency lookups under varying network sizes and churn rates with bounded routing table maintenance overhead. Furthermore, Accordion's automatic self-tuning algorithms approach the best possible performance/cost tradeoff, and Accordion's performance degrades only modestly when the node lifetimes do not follow the assumed Pareto distribution. Accordion stays within its bandwidth budget on average even when nodes have heterogeneous bandwidth budgets.

5.1 Experimental Setup

This evaluation uses an implementation of Accordion in p2psim, a publicly-available, discrete-event packet level simulator. Existing p2psim implementations of the Chord and OneHop DHTs simplified comparing Accordion to these protocols. The Chord implementation chooses neighbors based on their proximity [5, 7].

For simulations involving networks of less than 1740 nodes, we use a pairwise latency matrix derived from measuring the inter-node latencies of 1740 DNS servers using the King method [8]. However, because of the limited size of this topology and the difficulty involved in obtaining a larger measurement set, for simulations involving larger networks we assign each node a random 2D synthetic Euclidean coordinate and derive the network delay between a pair of nodes from their corresponding Euclidean distance. The average round-trip delay between node pairs in both the synthetic and measured delay matrices is 179 ms. Since each lookup for a random key starts and terminates at two random nodes, the average inter-host latency of the topology serves as a lower bound for the average DHT lookup latency. By default, our experiments use a Euclidean topology of 3000 nodes, except when noted. p2psim does not simulate link transmission rates or queuing delays. The experiments involve only key lookups; no data is retrieved.

Each node alternately leaves and re-joins the network; the interval between successive events for each node follows a Pareto distribution with median time of 1 hour (*i.e.*, $\alpha = 1$ and $\beta = 1800$ sec), unless noted. This choice of lifetime distribution is similar to past studies of peer-to-peer networks, as discussed in Section 3.3. Because $\alpha = 1$ in all simulations involving a Pareto distribution, our implementation of Accordion does not use the i_{thresh}-estimation technique presented in Section 4.6, as it is more convenient to set $i_{thresh} = p_{thresh} = .9$ instead.

Nodes issue lookups with respect to two different workloads. In the *churn intensive* workload, each node issues a lookup once every 10 minutes, while in the *lookup intensive* workload, each node issues a lookup once every 9 seconds. Experiments use the churn intensive workload unless otherwise noted. Each time a node joins, it uses a different IP address and DHT identifier. Each experiment runs for four hours of simulated time; statistics are collected only during the final half of the experiment and averaged over 5 simulation runs. All Accordion configurations set $b_{burst} = 100 r_{avg}$.

5.2 Comparison Framework

We evaluate the performance of the protocols using two types of metrics, *performance* and *cost*, following from the performance versus cost framework (PVC) we developed in previous work [16]. Though other techniques exist for comparing DHTs under churn [14, 17], PVC naturally allows us to measure how efficiently protocols achieve their performance vs. cost tradeoffs.

We measure performance as the average lookup latency

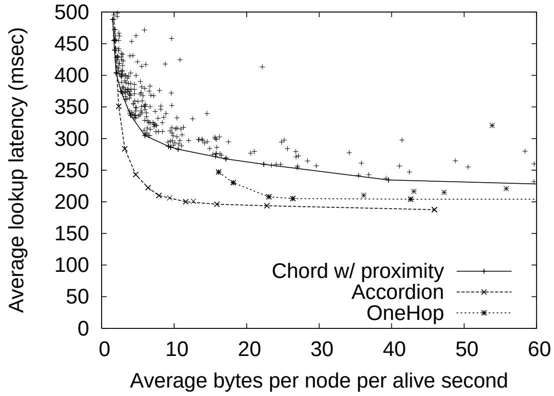

Figure 5: Accordion's bandwidth vs. lookup latency tradeoff compared to Chord and OneHop, using a 3000-node network and a churn intensive workload. Each point represents a particular parameter combination for the given protocol. Accordion's performance matches or improves OneHop's when bandwidth is plentiful, and Chord's when bandwidth is constrained.

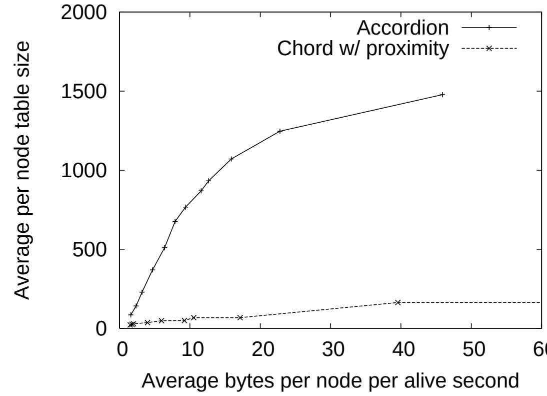

Figure 6: The average routing table size for Chord and Accordion as a function of the average per-node bandwidth, using a 3000-node network and a churn intensive workload. The routing table sizes for Chord correspond to the optimal parameter combinations in Figure 5. Accordion's ability to grow its routing table as available bandwidth increases explains why its latency is generally lower than Chord's.

of correct lookups (*i.e.*, lookups for which a correct answer is returned), including timeout penalties (three times the round-trip time to the dead node). All protocols retry failed lookups (*i.e.*, lookups that time out without completing) for up to a maximum of four seconds. We do not include the latencies of incorrect or failed lookups in this metric, but for all experiments of interest these counted for less than 5% of the total lookups for all protocols.

We measure cost as the average bandwidth consumed per node per alive second (*i.e.*, we divide the total bytes consumed by the sum of times that each node was alive). The size in bytes of each message is counted as 20 bytes for headers plus 4 bytes for each node mentioned in the message for Chord and OneHop. Each Accordion node entry is counted as 8 bytes due to additional fields on the bandwidth budget, node membership time (Δt_{alive}), and time since last contacted (Δt_{since}) for each node entry.

For graphs comparing DHTs with many parameters (*i.e.*, Chord and OneHop) to Accordion, we use PVC to explore the parameter space of Chord and OneHop fully and scatterplot the results. Each point on such a figure shows the average lookup latency and bandwidth overhead measured for one distinct set of parameter values for those protocols. The graphs also have the *convex hull* segments of the protocols, which show the best latency/bandwidth tradeoffs possible with the protocols, given the many different configurations possible. Accordion, on the other hand, has only one parameter, the bandwidth budget, and does not need to be explored in this manner.

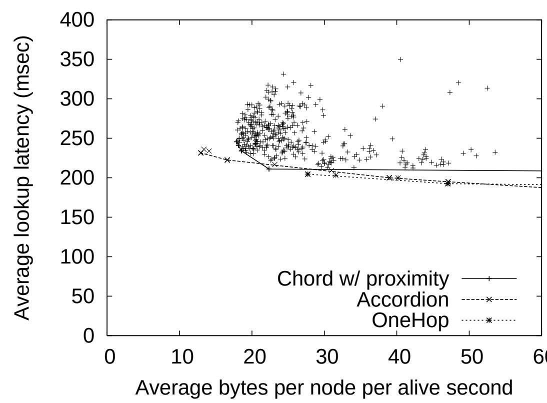

Figure 7: Accordion's lookup latency vs. bandwidth overhead tradeoff compared to Chord and OneHop, using a 1024-node network and a lookup intensive workload.

5.3 Latency vs. Bandwidth Tradeoff

A primary goal of the Accordion design is to adapt the routing table size to achieve the lowest latency depending on bandwidth budget and churn. Figure 5 plots the average lookup latency vs. bandwidth overhead tradeoffs of Accordion, Chord, and OneHop. In this experiment, we varied Accordion's r_{avg} parameter between 3 and 60 bytes per second. We plot measured actual bandwidth consumption, not the configured bandwidth budget, along the x-axis. The x-axis values include all traffic: lookups as well as routing table maintenance overhead.

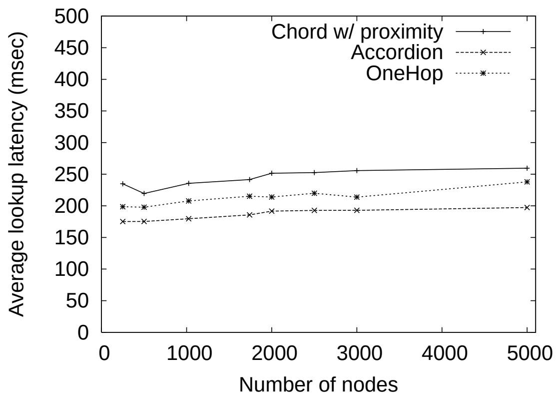

Figure 8: The lookup latency of Chord, Accordion and OneHop as the number of nodes in the system increases, using a churn intensive workload. Accordion uses a bandwidth budget of 6 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 7 and 23 bytes/node/sec in a 3000-node network, respectively.

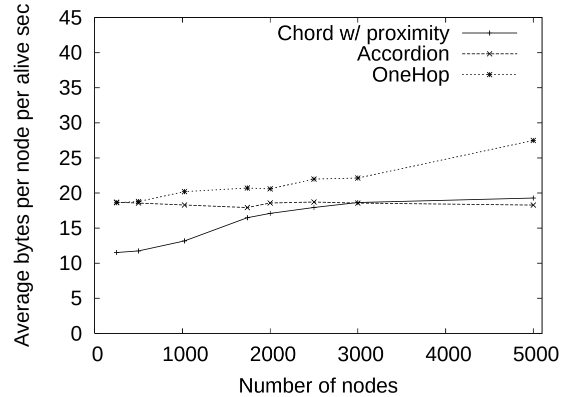

Figure 9: The average bytes consumed per node by Chord, Accordion and OneHop as the number of nodes in the system increases, from the same set of experiments as Figure 8.

Accordion approximates the lookup latency of the best OneHop configuration when the bandwidth budget is large, and the latency of the best Chord configuration when bandwidth is small. This is a result of Accordion's ability to adapt its routing table size, as illustrated in Figure 6. On the left, when the budget is limited, Accordion's table size is almost as small as Chord's. As the budgets grows, Accordion's routing table also grows, approaching the number of live nodes in the system (on average, half of the 3000 nodes are alive in the system).

As the protocols use more bandwidth, Chord cannot increase its routing table size as quickly as Accordion, even when optimally-tuned; instead, a node spends bandwidth on maintenance costs for its slowly-growing table. By increasing the table size more quickly, Accordion reduces the number of hops per lookup, and thus the average lookup latency.

Because OneHop keeps a complete routing table, all arrival and departure events must be propagated to all nodes in the system. This restriction prevents OneHop from being configured to consume very small amounts of bandwidth. As OneHop propagates these events more quickly, the routing tables are more up-to-date and both the expected hop count and timeouts per lookups decrease. Accordion, on the other hand, adapts its table size smoothly as its bandwidth budget allows, and can consistently maintain a fresher routing table, and thus lower latency lookups, than OneHop.

5.4 Effect of a Different Workload

The simulations in the previous section featured a workload that was *churn intensive*; that is, the amount of churn in the network was high in proportion to the lookup rate. This section evaluates the performance of Chord, OneHop, and Accordion under a *lookup intensive* workload. In this workload, each node issues one lookup every 9 seconds (almost 70 times more often than in the churn intensive workload), while the rate of churn is the same as that in the previous section.

Figure 7 shows the performance results for the three protocols. Again, convex hull segments and scatter plots characterize the performance of Chord and OneHop, while Accordion's latency/bandwidth curve is derived by varying the per-node bandwidth budget. As before, Accordion's performance approximates OneHop's when bandwidth is high.

In contrast to the churn intensive workload, in the lookup intensive workload Accordion can operate at lower levels of bandwidth consumption than Chord. With a low lookup rate as in Figure 5, Chord can be configured with a small base (and thus small routing table and more lookup hops, accordingly) to achieve low latencies, with relatively high lookup latencies. However, with a high lookup rate as in Figure 7, using a small base in Chord is not the best configuration: it has relatively high lookup latency, but also has a large overhead due to the large number of forwarded lookups. Because Accordion learns new routing entries from lookup traffic, a higher rate of lookups leads to a larger per-node routing table, resulting in fewer lookup hops and less overhead due to forwarding lookups. Thus, Accordion can operate at lower levels of bandwidth than Chord because it automatically increases its routing table size by learning from the large number of lookups.

The rest of the evaluation focuses on the churn intensive workload, unless otherwise specified.

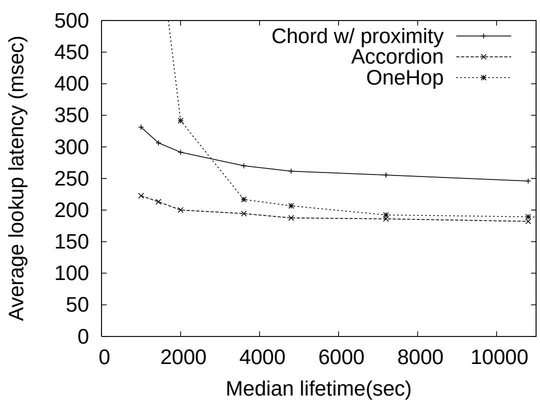

Figure 10: The lookup latency of Chord, Accordion and OneHop as median node lifetime increases (and churn decreases), using a 3000-node network. Accordion uses a bandwidth budget of 24 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 17 and 23 bytes/node/sec, respectively, with median lifetimes of 3600 sec.

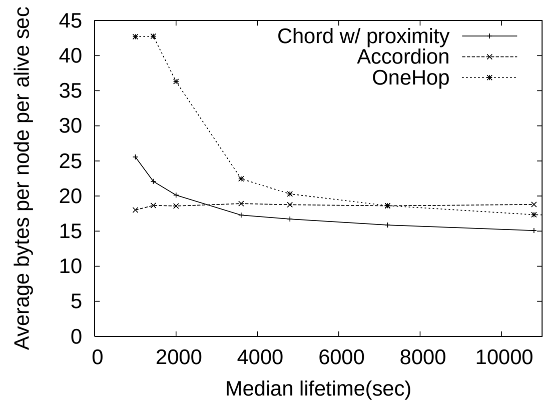

Figure 11: The average bytes consumed per node by Chord, Accordion and OneHop as median node lifetime increases (and churn decreases), from the same set of experiments as Figure 10.

5.5 Effect of Network Size

This section investigates the effect of scaling the size of the network on the performance of Accordion. Figures 8 and 9 show the average lookup latency and bandwidth consumption of Chord, Accordion and OneHop as a function of the network size. For Chord and OneHop, we fix the protocol parameters to be the optimal settings in a 3000-node network (*i.e.*, the parameter combinations that produce latency/overhead points lying on the convex hull segments) for bandwidth consumptions of 17 bytes/node/sec and 23 bytes/node/sec, respectively. For Accordion, we fix the bandwidth budget at 24 bytes/sec. With fixed parameter settings, Figure 9 shows that both Chord and OneHop incur increasing overhead that scales as $\log n$ and n respectively, where n is the size of the network. However, Accordion's fixed bandwidth budget results in predictable overhead consumption regardless of the network size. Despite using less bandwidth than OneHop and the fact that Chord's bandwidth consumption approaches that of Accordion as the network grows, Accordion's average lookup latency is consistently lower than that of both Chord and OneHop.

These figures plot the *average* bandwidth consumed by the protocols, which hides the bandwidth that is consumed on per-node or burst levels. Because Accordion controls bandwidth bursts, it keeps individual nodes within their bandwidth budgets. OneHop, however, explicitly distributes bandwidth unevenly: slice leaders [9] typically use 7 to 10 times the bandwidth of average nodes. OneHop is also more bursty than Accordion; we observe that the maximum bandwidth burst observed for OneHop is 1200 bytes/node/sec in a 3000-node network, more than 10 times the maximum burst of Accordion. Thus, OneHop's bandwidth consumption varies widely and could at any one time exceed a node's desired bandwidth budget, while Accordion stays closer to its average bandwidth consumption.

5.6 Effect of Churn

Previous sections illustrated Accordion's ability to adapt to different bandwidth budgets and network sizes; this section evaluates its adaptability to different levels of churn.

Figures 10 and 11 shows the lookup latency and bandwidth overhead of Chord, Accordion and OneHop as a function of median node lifetime. Lower node lifetimes correspond to higher churn. Accordion's bandwidth budget is constant at 24 bytes per second per node. Chord and OneHop uses parameters that achieve the lowest lookup latency while consuming 17 and 23 bytes per second, respectively, for a median node lifetime of one hour. While Accordion maintains fixed bandwidth consumption regardless of churn, both Chord and OneHop's overhead grow inversely proportional to median node lifetime (proportional to churn rates). Accordion's average lookup latency increases with shorter median node lifetimes, as it maintains a smaller table due to higher eviction rates under high churn. Chord's lookup latency increases due to a larger number of lookup timeouts, because of its fixed table stabilization interval. Accordion's lookup latency decreases slightly as the network becomes more stable, with consistently lower latencies than both Chord and OneHop. OneHop has unusually high lookup latencies under high churn as its optimal setting for the event aggregation interval with mean node lifetimes of 1 hour is not ideal under higher churn, and as a result lookups incur more frequent timeouts due to stale routing table entries.

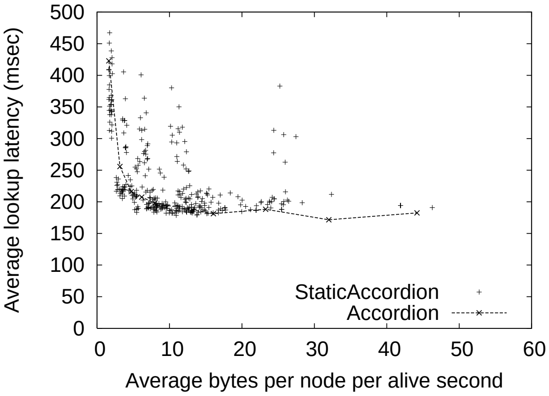

Figure 12: Bandwidth versus latency for Accordion and StaticAccordion, using a 1024-node network and a churn intensive workload. Accordion tunes itself nearly as well as the best exhaustive-search parameter choices for StaticAccordion.

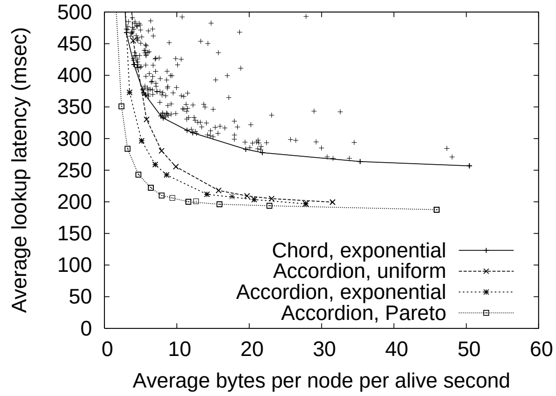

Figure 13: The performance of Accordion on three different node lifetime distributions, and of Chord on an exponential distribution, using a 3000-node network and a churn intensive workload. Though Accordion works best with a Pareto distribution, it still outperforms Chord with an exponential node lifetime distribution in most cases.

Parameter	**Range**
Exploration interval	2-90 sec
Lookup parallelism w_p	1,2,4,6
Eviction threshold i_{thresh}	.6 –.99

Table 1: StaticAccordion parameters and ranges.

5.7 Effectiveness of Self-Tuning

Accordion adapts to the current churn and lookup rate by adjusting w_p and the frequency of exploration, in order to stay within its bandwidth budget. To evaluate the quality of the adjustment algorithms, we compare Accordion with a simplified version (called StaticAccordion) that uses fixed w_p, i_{thresh} and active exploration interval parameters. Simulating StaticAccordion with a range of parameters, and looking for the best latency vs. bandwidth tradeoffs, indicates how well Accordion could perform with ideal parameter settings. Table 1 summarizes StaticAccordion's parameters and the ranges explored.

Figure 12 plots the latency vs. bandwidth tradeoffs of StaticAccordion for various parameter combinations. The churn and lookup rates are the same as the scenario in Figure 5. The lowest StaticAccordion points, and those farthest to the left, represent the performance Accordion could achieve if it self-tuned its parameters optimally. Accordion approaches the best static tradeoff points, but has higher latencies in general for the same bandwidth consumption. This is because Accordion tries to control bandwidth overhead, such that it not exceed the maximum-allowed burst size if possible (where we let $b_{burst} = 100 r_{avg}$). StaticAccordion, on the other hand, does not attempt to regulate its burst size. For example, when the level of lookup parallelism is high, a burst of lookups will generate a large burst of traffic. However, Accordion will reduce the lookup parallelism w_p to try to stay with the maximum burst size. Therefore, StaticAccordion can keep its lookup parallelism constant to achieve lower latencies (by masking more timeouts) than Accordion, though the average bandwidth consumption will be the same in both cases. As such, if controlling bursty bandwidth is a goal of the DHT application developer, Accordion will control node bandwidth more consistently than StaticAccordion, without significant additional lookup latency.

5.8 Lifetime Distribution Assumption

Accordion's algorithm for predicting neighbor liveness probability assumes a heavy-tailed Pareto distribution of node lifetimes (see Sections 3.3 and 4.6). In such a distribution, nodes that have been alive a long time are likely to remain alive. Accordion exploits this property by preferring to keep long-lived nodes in the routing table. If the distribution of lifetimes is not what Accordion expects, it may make more mistakes about which nodes to keep, and thus suffer more lookup timeouts. This section evaluates the effect of such mistakes on lookup latency.

Figure 13 shows the latency/bandwidth tradeoff with node lifetime distributions that are uniform and exponential. The uniform distribution chooses lifetimes uniformly at random between six minutes and nearly two hours, with an average of one hour. In this distribution, nodes that have been part of the network longer are *more* likely to fail soon. In the exponential distribution, node lifetimes are exponentially distributed with a mean of one hour; the probability of a node being alive does not depend on its join time.

Figure 13 shows that Accordion's lookup latencies are

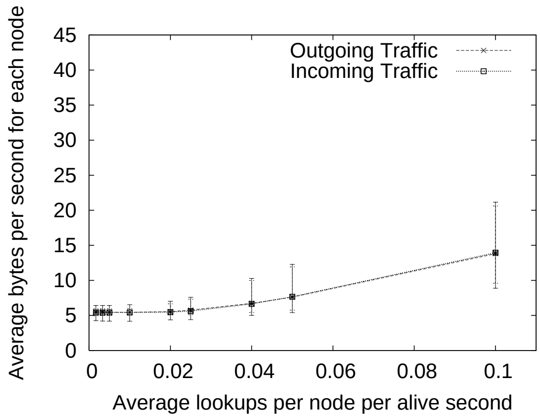

Figure 14: Accordion's bandwidth consumption vs. lookup rate, using a 3000-node network and median node lifetimes of one hour. All nodes have a bandwidth budget of 6 bytes/sec. Nodes stay within the budget until the lookup traffic exceeds that budget.

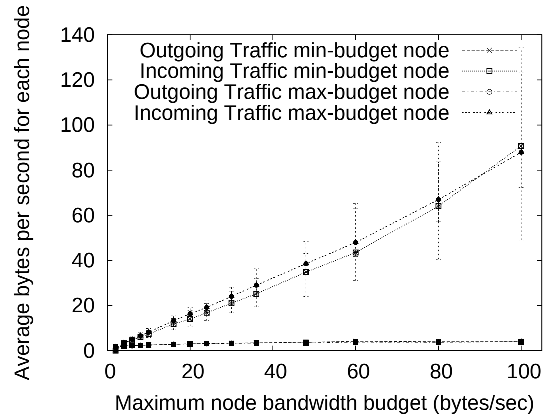

Figure 15: Bandwidth consumption of Accordion nodes in a 3000-network using a churn intensive workload where nodes have heterogeneous bandwidth budgets, as a function of the largest node's budget. For each experiment, nodes have budgets uniformly distributed between 2 and the x-value. This figure shows the consumption of the nodes with both the minimum and the maximum budgets.

higher with uniform and exponential distributions than they are with Pareto. However, Accordion still provides lower lookup latencies than Chord, except when bandwidth is very limited.

5.9 Bandwidth Control

An Accordion node does not have direct control over all of the network traffic it generates and receives, and thus does not always keep within its bandwidth budget. A node must always forward primary lookups, and must acknowledge all exploration packets and lookup requests in order to avoid appearing to be dead. This section evaluates how much Accordion exceeds its budget.

Figure 14 plots bandwidth consumed by Accordion as a function of lookup traffic rate, when all Accordion nodes have a bandwidth budget of 6 bytes/sec. The figure shows the median of the per-node averages over the life of the experiment, along with the 10^{th} and 90^{th} percentiles, for both incoming and outgoing traffic. When lookup traffic is low, nodes achieve exactly 6 bytes/sec. As the rate of lookups increases, nodes explore less often and issue fewer parallel lookups. Once the lookup rate exceeds one every 25 seconds there is too much lookup traffic to fit within the bandwidth budget. Each lookup packet and its acknowledgment cost approximately 50 bytes in our simulator, and our experiments show that at high lookup rates, lookups take nearly 3.6 hops on average (including the direct reply to the query source). Thus, for lookup rates higher than 0.04 lookups per second, we expect lookup traffic to consume more than $50 \cdot 3.6 \cdot 0.04 = 7.2$ bytes per node per second, leading to the observed increase in bandwidth.

The nodes in Figure 14 all have the same bandwidth budget. If different nodes have different bandwidth budgets, it might be the case that nodes with large budgets force low-budget nodes to exceed their budgets. Accordion addresses this issue by explicitly biasing lookup and exploration traffic towards neighbors with high budgets. Figure 15 shows the relationship between the spread of budgets and the actual incoming and outgoing bandwidth incurred by the lowest- and highest-budget nodes. The node budgets are uniformly spread over the range $[2, x]$ where x is the maximum budget shown on the x-axis of Figure 15. Figure 15 shows that the bandwidth used by the lowest-budget node grows very slowly with the maximum budget in the system; even when there is a factor of 50 difference between the highest and lowest budgets, the lowest-budget node exceeds its budget only by a factor of 2. The node with the maximum budget stays within its budget on average in all cases.

6 Related Work

Unlike other DHTs, Accordion is not based on a particular data structure and as a result it has great freedom in choosing the size and content of its routing table. The only constraint it has is that the neighbor identifiers adhere to the small-world distribution [13]. Accordion has borrowed routing table maintenance techniques, lookup techniques, and inspiration from a number of DHTs [9–11, 20, 23, 25], and shares specific goals with MSPastry, EpiChord, Bamboo, and Symphony.

Castro et al. [2] present a version of Pastry, MSPastry, that self-tunes its stabilization period to adapt to churn and achieve low bandwidth. MSPastry also estimates the current failure rate of nodes, using historical failure observations. Accordion shares the goal of automatic tuning, but focuses on adjusting its table size as well as adapting the rate of maintenance traffic.

Instead of obtaining new state by explicitly issuing lookups for appropriate identifiers, Accordion learns information from the routing tables of its neighbors. This form of information propagation is similar to classic epidemic algorithms [6]. EpiChord [15] also relies on epidemic propagation to learn new routing entries. EpiChord uses parallel iterative lookups, as opposed to the parallel recursive lookups of Accordion, and therefore is not able to learn from its lookup traffic according to the identifier distribution of its routing table.

Bamboo [22], like Accordion, has a careful routing table maintenance strategy that is sensitive to bandwidth-limited environments. The authors advocate a fixed-period recovery algorithm, as opposed to the more traditional method of recovering from neighbor failures reactively, to cope with high churn. Accordion uses an alternate strategy of actively requesting new routing information only when bandwidth allows. Bamboo also uses a lookup algorithm that attempts to minimize the effect of timeouts, through careful timeout tuning. Accordion avoids timeouts by predicting the liveness of neighbors and using parallel lookups.

Symphony [19] is a DHT protocol that also uses a small-world distribution for populating its routing table. While Accordion automatically adjusts its table size based on a user-specified bandwidth budget and churn, the size of Symphony's routing table is a protocol parameter. Symphony acquires the desired neighbor entries by explicitly looking up identifiers according to a small-world distribution. Accordion, on the other hand, acquires new entries by learning from existing neighbors during normal lookups and active exploration. Existing evaluations of Symphony [19] do not explicitly account for bandwidth consumption nor the lookup latency penalty due to timeouts. Mercury [1] also employs a small-world distribution for choosing neighbor links, but optimizes its tables to handle scalable range queries rather than single key lookups.

A number of file-sharing peer-to-peer applications allow the user to specify a maximum bandwidth. Gia [3] exploits that information to explicitly control the bandwidth usage of nodes by using a token-passing scheme to approximate flow control.

7 Conclusion

We have presented Accordion, a DHT protocol with a unique design that automatically adjusts itself to reflect current operating environments and a user-specified bandwidth budget. By learning about new routing state opportunistically through lookups and active search, and evicting state based on liveness probability estimates, Accordion adapts its routing table size to achieve low lookup latency while staying within a user-specified bandwidth budget.

A self-tuning, bandwidth-efficient protocol such as Accordion has several benefits. Users often don't have the expertise to tune every DHT parameter correctly for a given operating environment; by providing them with a single, intuitive parameter (a bandwidth budget), Accordion shifts the burden of tuning from the user to the system. Furthermore, by remaining flexible in its choice of routing table size and content, Accordion can operate efficiently in a wide range of operating environments, making it suitable for use by developers who do not want to limit their applications to a particular network size, churn rate, or lookup workload.

Currently, we are instrumenting DHash [5] to use Accordion. Our p2psim version of Accordion is available at: `http://pdos.lcs.mit.edu/p2psim`.

Acknowledgments

We thank Frank Dabek for the King dataset measurements, Russ Cox and Thomer Gil for their help writing the simulator, and Anjali Gupta for implementing OneHop in p2psim. We are grateful to David Karger, Max Krohn, Athicha Muthitacharoen, Emil Sit, the OceanStore group at UC Berkeley, and the anonymous reviewers for their insightful comments on previous drafts of this paper. Our shepherd, Miguel Castro, provided valuable feedback that helped improve this paper.

References

[1] Bharambe, A. R., Agrawal, M., and Seshan, S. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the 2004 SIGCOMM* (Aug. 2004).

[2] Castro, M., Costa, M., and Rowstron, A. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 DSN* (June 2004).

[3] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. Making Gnutella-like P2P systems scalable. In *Proceedings of the 2003 SIGCOMM* (August 2003).

[4] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: A decentralized network coordinate system. In *Proceedings of the 2004 SIGCOMM* (Aug. 2004).

[5] Dabek, F., Li, J., Sit, E., Robertson, J., Kaashoek, M. F., and Morris, R. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st NSDI* (March 2004).

[6] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th PODC* (Aug. 1987).

[7] Gummadi, K. P., Gummadi, R., Gribble, S. D., Ratnasamy, S., Shenker, S., and Stoica, I. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 SIGCOMM* (Aug. 2003).

[8] GUMMADI, K. P., SAROIU, S., AND GRIBBLE, S. D. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop* (Nov. 2002).

[9] GUPTA, A., LISKOV, B., AND RODRIGUES, R. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st NSDI* (Mar. 2004).

[10] GUPTA, I., BIRMAN, K., LINGA, P., DEMERS, A., AND VAN RENESSE, R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd IPTPS* (Feb. 2003).

[11] KAASHOEK, M. F., AND KARGER, D. R. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd IPTPS* (Feb. 2003).

[12] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th STOC* (May 1997).

[13] KLEINBERG, J. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd STOC* (May 2000).

[14] KRISHNAMURTHY, S., EL-ANSARY, S., AURELL, E., AND HARIDI, S. A statistical theory of chord under churn. In *Proceedings of the 4th IPTPS* (Feb. 2005).

[15] LEONG, B., LISKOV, B., AND DEMAINE, E. D. Epichord: Parallelizing the Chord lookup algorithm with reactive routing state management. In *Proceedings of the 12th International Conference on Networks* (Nov. 2004).

[16] LI, J., STRIBLING, J., MORRIS, R., KAASHOEK, M. F., AND GIL, T. M. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the 24th INFOCOM* (Mar. 2005).

[17] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. R. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st PODC* (Aug. 2002).

[18] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems 21*, 4 (Dec. 1996), 480–525.

[19] MANKU, G. S., BAWA, M., AND RAGHAVAN, P. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[20] MAYMOUNKOV, P., AND MAZIÈRES, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st IPTPS* (Mar. 2002).

[21] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content addressable network. In *Proceedings of the 2001 SIGCOMM* (Aug. 2001).

[22] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference* (June 2004).

[23] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[24] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems Journal 9*, 2 (Aug. 2003), 170–184.

[25] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking 11*, 1 (Feb. 2003), 149–160.

[26] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004), 41–53.