Local Load Balancing in Distributed Hash Tables

Chris Riley*
Johns Hopkins University
chrisr@cs.jhu.edu

Christian Scheideler
Johns Hopkins University
scheideler@cs.jhu.edu

Abstract

There are many existing peer-to-peer systems for distributed hash tables in which both nodes and tokens are mapped to a shared identifier space and tokens are assigned to nearby (often successor) nodes. The load balancing performance of this straightforward mapping is limited, since nodes may be mapped very near each other. Virtual nodes and other techniques have been used to correct this, but existing techniques reduce scalability by increasing node degree significantly or by requiring multiple independent lookup operations for a single token. We present three algorithms based on local load balancing which improve the load distribution without such cost. The first is a local load balancing process in which each node simply balances with its neighbors; it can balance the distribution to a maximum load of $O(\frac{1}{n})$ in a single round of improvements in hypercubic structures. The other two algorithms modify the identifier mapping process to balance token assignments, improving the load distribution to a maximum load of $O(\frac{1}{n})$ while reducing the worst-case node degree and maintaining a low expected network congestion.

*Contact author. Address: Dept. of Computer Science, Johns Hopkins University, 3400 N. Charles Street, Baltimore, MD 21218, USA. Phone: 410-516-4650. Fax: 410-516-6134.

1 Introduction

Distributed hash tables, structures which assign tokens to a set of nodes without maintaining a central table, are common in peer-to-peer systems research. The tokens can be used to represent data objects for storage or computational tasks, and are generally considered to be of identical size. The nodes are highly dynamic, and in most cases are considered to be identical in capacity. The hash table must connect the nodes in a way that handles node joins and leaves and allows tokens to be quickly located and distributed fairly.

Many existing systems (such as Chord[13]) are built by assigning nodes and tokens to points in the same identifier space, usually the $[0, 1)$ real line connected into a circle, and mapping each token to the node with the nearest point or the nearest succeeding point in the space (Figure 1). This approach is less than ideal in its load balancing; some nodes can receive a $O(\frac{\log n}{n})$ fraction of the space in a system of n nodes. To counter this, virtual nodes can be used, mapping each node to $O(\log n)$ points in the space. But this approach causes the degree of each node to increase by a $O(\log n)$ factor, which limits scalability.

An alternate to using virtual nodes is to implement a separate load balancing strategy on top of the distributed hash table. Since no node is aware of the entire system at any time, local load balancing stategies must be used, strategies which allow each node to balance itself with a few other nodes considered nearby. This resembles the local load balancing that occurs in parallel computing, but there are two qualifications here that are not often considered in parallel computing. First, the set of participants is highly dynamic. Second, the tokens must be efficiently retrievable. It is also desirable for a network to have a balanced degree and low congestion. Load balancing algorithms often try to balance the structure of the network, and extensions of these could be considered to remap edges to produce a more uniform degree and a lower network congestion.

We present three local load balancing algorithms for extending distributed hash tables based on a shared identifier space. One algorithm, *neighbor balancing*, moves tokens to neighboring nodes and works wth any pointer structure, and can improve hypercubic structures to an $O(\frac{1}{n})$ distribution with a single round of token movements. The *point remapping* algorithm adjusts the location of nodes in the $[0, 1)$ circle to balance the size of ranges; the *range sharing* algorithm maps each token and edge endpoint to a random node near the originally assigned node. Both of these algorithms produce distributions with maximum expected load $O(\frac{1}{n})$, and produce structures with low degree and congestion.

1.1 Related work

Distributed hash tables are often based on the technique of consistent hashing[7] or the work of Plaxton *et al*[9]. Common distributed hash tables include Chord[13], Koorde[6], Distance Halving[8], Pastry[12], Tapestry[5], and CAN[11], to name a few. We do not replace such systems, but suggest the use of our strategies in addition to them.

To perform load balancing, many distributed hash tables use virtual nodes, creating $O(\log n)$ virtual nodes (separate points on the identifier circle) for each node in the system and mapping tokens to the virtual nodes, collecting them at the physical nodes; see for example [10]. This strategy can give a nearly uniform load balance, though it increases node degree and other costs by a factor of $O(\log n)$, which in many systems limits scalability.

Byers *et al.* [3] consider instead the application of multiple-choice hashing techniques to load balancing in distributed hash tables, mapping each token to multiple points in the identifier circle and assigning the token to the least loaded successor of a point. This causes lookup costs to increase, though they use redirection pointers to reduce this cost somewhat.

Adler *et al.* [1] give a hypercubic hash table similar to the approach used by CAN[11]. Their table assigns nodes to finite binary strings and hashes tokens to $[0, 1)$, mapping tokens to the node whose binary string is a prefix of the binary decimal representation of the token's hash. They view the set of nodes as leaves in an incomplete binary tree and maintain a constant difference in depth between the highest and lowest leaf, thus assigning an expected $O(\frac{1}{n})$ share of the load to every node. Their work considers only a sequence of n insertions.

Local load balancing has been extensively studied in the context of load balancing in parallel computers. There are numerous papers and textbooks on the subject; for an example, see [14]. Ghosh *et al.*[4] analyze convergence time and performance for a local load balancing algorithm in dynamic and asynchronous networks, but their algorithm does not take into account efficient token retrieval and therefore is not practical for general-purpose DHT's.

1.2 Assumptions

We assume that node capacities are *uniform*, in the sense that each of n nodes should be given a $\frac{1}{n}$ share of the tokens; most existing work deals with uniform capacity nodes.

We do not consider reliability in the sense that we do not attempt to make token storage persistent across node failures. As with existing systems, it is trivial to create multiple copies of a token and map them to independent random points on the circle for redundancy; then each token is available with high probability against stochastic failures if $O(\log n)$ copies are used and nodes are available with constant probability.

1.3 Our results

The first method we propose, *neighbor balancing*, allows each node to redistribute some tokens assigned to it by the original DHT to its neighbors in the DHT's data structure; this strategy can be used in any data structure without modification but requires a convergence process. The second method, *point remapping*, adjusts the initial point in the identifier circle assigned by the DHT to make the ranges more uniform in size. The third method, *local sharing*, assigns tokens randomly to a set of adjacent nodes through a local decision process at the original node assigned by the DHT.

The performance for each algorithm is as follows:

1. The neighbor balancing algorithm converges to a distribution with optimal maximum node load over any local rebalancing of the load under the same constraints, regardless of the topology of the underlying system. Furthermore, in a system with hypercubic connections this distribution has a constant factor load balance.

2. When using the point remapping algorithm, every node has a range of size $O(\frac{1}{n})$ with high probability.

3. When using the range sharing algorithm, each node receives a $O(\frac{1}{n})$ share of the tokens in the system with high probability.

The cost of the algorithms is:

1. In the neighbor balancing algorithm, some additional tokens must be moved to process node join and leave operations, but no additional nodes are contacted. Token lookups require each neighbor of the initial node to be contacted, which in all existing systems is at most a factor of 2 overhead.

2. In the point remapping algorithm, $O(\log n)$ nodes are displaced with each node join and leave, and their neighbors may need to be contacted; in a hypercubic graph, for example, $O(\log^2 n)$ nodes must be contacted for each operation. Many tokens may need to be moved as well. The cost of token lookups is unchanged.

3. In the range sharing algorithm, with high probability each node has $O(\log n)$ additional neighbors which must be contacted with each join/leave, and must keep $O(\log^2 n)$ additional local storage. An efficient implementation can keep token movements during node joins and leaves to an optimum.

While point remapping seems worse than the other strategies, it is the only algorithm which balances node usage for token lookup, since to find a token the other two strategies first must contact the node selected by the initial unbalanced distribution.

The neighbor balancing algorithm cannot improve degree or congestion since it makes no structural modifications. The other two can improve these. We consider the original structure connections to be based on a set of d relative ideal edges; for example, in Chord, the relative ideal edges are $\frac{1}{2^i}$ for i from 1 to $\log n$. We analyze congestion based on the random routing problem in a hypercubic structure (for simplicity).

1. Point remapping reduces total node degree to worst-case $O(d)$. In hypercubic structures, the expected congestion is $O(\log n)$.

2. Range sharing reduces total node degree to worst-case $O(d + \log n)$ with high probability. In hypercubic structures the expected congestion is $O(\log n)$.

2 Neighbor balancing

2.1 Algorithm

In the neighbor balancing algorithm, the data structure used by the original DHT is left unchanged, but each node is allowed to move some of the tokens assigned to it to its neighbors; tokens are restricted to be at most one step away from the originally assigned node. This is the most flexible of the algorithms since it requires the least modification to the system and can be used in any pointer structure. Unlike the other algorithms, though, there is an initial imbalance which must converge to a balanced state. We present the algorithm and compare it to an optimal offline algorithm that is given the same network and initial distribution and can only move tokens one node away from the original node.

A node v can perform two types of balancing:

1. One-hop: v moves tokens initially assigned to it from itself to a neighbor

2. Two-hop: v moves tokens initially assigned to it from one neighbor to another

We propose one solution for the token movement and balance algorithms, but other algorithms could be used. We propose a memoryless token movement, simply shifting the token to the neighbor and keeping a count of the number of tokens held by each neighbor (Figure 2). We balance the load using pairwise load averages. Take any two nodes u and v where $l(u) > l(v)$ and u has tokens that can be moved to v (through one or two hops); send tokens so that $l'(u) = l'(v) = \frac{l(u)+l(v)}{2}$ or until u cannot send any more tokens to v. We require u and v to be involved in only one balance operation at a time.

2.2 Cost

The only additional token movements are for a departing node returning tokens held for neighbors or for a joining node accepting tokens from neighbors. Since we only allow tokens to move one hop away from their original assigned state, the only contacts to exchange tokens for a join or leave are neighbors, which must be contacted anyway. Token lookups may require each neighbor to be contacted, which is not a significant overhead.

2.3 Performance

We begin by showing that the algorithm converges to an optimal state given the input graph and token distribution and the requirement that tokens be moved no further than one edge away from their initial target. We then analyze the load balance of such a state for a hypercubic input graph and consistent-hashing derived initial distribution. In this section, a *stable state* for a fixed network and given initial load distribution is a local rearrangement of tokens in which neither one-hop nor two-hop rebalances can improve the load.

Lemma 2.1 *Every stable state has optimal max load, or the maximum node load in a stable state is minimum over all local-only rebalancings of the given network and initial load distribution.*

Proof. Consider a connected maximal set S of nodes all of maximum load in the system; to make the problem non-trivial we assume $S \neq V$. Let $B(S)$ be all nodes on the boundary of the set, the nodes in S which have an edge to nodes outside S, and let $I(S) = S - B(S)$ be the interior of S. Since no one-hop or two-hop rebalances can be performed, and since each node in $B(S)$ has a neighbor outside S which is not of maximum load, all load placed in $B(S)$ in the initial distribution is outside S in the stable state; if not, then it would either be in the original node or in some other node adjacent to the original node, and it would be moved outside S with a one-hop or two-hop rebalancing operation. Further, the nodes in $B(S)$ are not holding any load from outside S. Therefore all load held by $B(S)$ was originally assigned to $I(S)$.

Since every one-hop rebalancing must place the load initially assigned to $I(S)$ within S, and since in any stable state this load is evenly placed within S, no rebalancing can produce a lower maximum load. □

Lemma 2.2 *The neighbor balancing algorithm converges to a stable state.*

Proof. Suppose there were a cyclical sequence of operations, returning the load distribution to a previous state. Consider any node v modified by the cycle which reaches a minimum load over the load at all times of all nodes modified by the cycle. Then at some time in the cycle, v's load is reduced to this minimum load; this implies that it was balanced with a node w with less than v's initial minimum load, which is a contradiction.

Since there are no cycles, the system continues to make progress with each rebalancing operation. Since in every time step at least one token can be moved, eventually the system will be unable to make progress. Since we use load averaging, it is impossible for the system to be unable to make progress without being at a stable state. □

Theorem 2.3 *In a fixed network, the neighbor balancing algorithm converges to a distribution with optimal maximum node load.*

Proof. Follows from Lemmas 2.2 and 2.1. □

But how good is an optimal state under our restrictions? We begin with a supporting lemma:

Lemma 2.4 *For any set S of nodes with $|S| \geq C \log n$ for some constant C, with high probability the total load assigned to all nodes of S in a random assignment of n nodes and T tokens to the identifier circle is $\Theta(|S| \cdot \frac{T}{n})$.*

Proof. The proof involves two steps, bounding the size of the range held by S and then using Chernoff bounds for the number of tokens assigned to the range. We omit details since the proof follows from analysis in [7]. □

Theorem 2.5 *With high probability any stable rebalancing of a random assignment of nodes and tokens to a distributed hash table with hypercubic connections has a constant factor load balance.*

Proof. Consider a connected maximal set S of nodes of maximum load in any stable state, and divide S into interior nodes $I(S)$ without neighbors outside S and boundary nodes $B(S)$ with neighbors outside S. Consider the case $|I(S)| \geq C \log n$ for the constant C in Lemma 2.4. Then $|S| > C \log n$, and the total load assigned to S is $O(|S| \cdot \frac{T}{n})$. Since by definition the load in S is evenly distributed across all nodes, and since S only holds load assigned to $I(S) \subset S$, each node has load $O(\frac{T}{n})$. Suppose instead $|I(S)| < C \log n$. Then, for sufficiently large n, $|B(S)| = \Omega(|I(S)| \cdot \log n)$ since the network is hypercubic and $B(S)$ contains all neighbors of $I(S)$ by definition. With high probability the most load that can be in a single node is $O(\frac{T \log n}{n})$, so that the most load in $I(S)$ even for small $I(S)$ is $O(\frac{T \log n |I(S)|}{n})$ with high probability. Since this load is evenly distributed across the nodes in S, the load per node is $O(\frac{T}{n})$. □

2.4 Faster convergence

Other algorithms for balancing can speed the convergence process, including algorithms which take node degree into account and balance with many neighbors in parallel, and algorithms which force a large node to balance with its smallest neighbor possible. Consider a parallel degree-biased algorithm. Each node v with degree $d(v)$ and load $l(v)$ sends to each $\{u : u \in N(v), l(v) > l(u)\}$ load $\frac{1}{d(v)}(l(v) - l(u))$, or as much as v can send to u (where v can send to u any load initially assigned to u held by v and any load assigned to v and held by v).

Theorem 2.6 *In a graph with hypercubic connections and a consistent-hashing based initial assignment the degree-biased algorithm produces a constant factor load distribution with a single round of balancing with high probability.*

Proof. The original consistent hashing paper showed that with high probability no node receives a greater than $\frac{\log n}{n}$ fraction of the system load[7]. Since each node has $\log n$ neighbors, if $\epsilon \log n$ of them have load $O(\frac{1}{n})$ for any constant $\epsilon > 0$, then the theorem holds.

Lemma 2.4 says that for any set S with $|S| \geq C \log n$ for some constant C, the load assigned to set S is a $\Theta(\frac{|S|}{n})$ fraction of the total system load. Any node v and its neighborhood $N(v)$ represents a set of size $(\log n + 1)$; since this set has fewer than $C \log n$ nodes, the set is assigned at most a $O(\frac{\log n}{n})$ fraction of the system load. If $N(v)$ does not contain $\epsilon \log n$ nodes with constant load for any ϵ, then $\log n - o(\log n)$ nodes have load $\omega(\frac{1}{n})$, which implies that the set collectively has load $\omega(\frac{\log n}{n})$, which is a contradiction. □

2.5 Degree and congestion balancing

Since no structural modifications are made, this algorithm cannot reduce the degree or congestion.

3 Point remapping

3.1 Overview

The *point remapping* algorithm modifies the original process for node insertion. After a node is inserted into the system, it moves some of the nodes around it so that they are evenly spaced. More specifically, the newly inserted node v is placed at its hash value and becomes a *fixed node* (a node which is not a fixed node is a *movable node*). It finds two other fixed nodes around the circle at appropriate distances, one in each direction, and balances the placements of the other nodes between these end nodes (Figure 3).

A correct mapping of nodes to points satisfies the following:

Invariant 3.1 *The ordering of nodes determined by their originally mapped points is preserved.*

Invariant 3.2 *Fixed nodes are at least $c_1 \log n$ and at most $c_2 \log n$ nodes apart for some constants c_1 and c_2.*

Invariant 3.3 *Between any two adjacent fixed nodes all distances between nodes are identical.*

3.2 Algorithm

We present the insertion of a single node assuming that before the insertion the invariants hold. We require each node to store its original location on the circle. We also require that c_1 be at least the constant in Lemma 2.4 and $c_2 > 2c_1$. Furthermore, we assume each node is connected to its left and right neighbors around the circle.

As stated above, the newly inserted node v becomes a fixed node located at its assigned point on the circle. It contacts nodes to its left and right until it finds the left and right fixed neighbors l_f and r_f. This requires $O(\log n)$ steps since fixed nodes are at most $c_2 \log n$ apart. The algorithm has two phases:

Phase 1: Check to see if l_f or r_f is too close to v, where $d(v, l_f)$ is the number of nodes which have original locations between those of v and l_f; by Invariant 3.1, all such nodes lie between l_f and r_f. If $d(v, l_f) < c_1 \log n$ (or r_f similarly), then l_f becomes a movable node. Find the next fixed node beyond l_f or go $c_2 \log n$ nodes away from v, whichever comes first. If another fixed node is found, it becomes the new l_f. If not, the node $\frac{c_2 \log n}{2}$ steps away from v is selected to be the new l_f and is moved to its original location.

Phase 2: Once v and l_f and r_f are chosen and placed at their original locations, the nodes with original locations between l_f (resp. r_f) and v are evenly spaced between l_f (resp. r_f) and v, preserving their ordering.

Lemma 3.4 *The algorithm preserves the invariants.*

Proof. Clearly Invariant 3.3 and Invariant 3.1 are met by the second phase of the algorithm. Neither l_f nor r_f as originally selected can be too far away from v, since it is between them and they are not too far apart. Suppose without loss of generality that l_f was too close to v, and a nearby fixed node is used instead. By definition it is not too far from v, and it could not be too close since it would have been too close to l_f. Suppose instead that no nearby fixed node was found. Then the new l_f as selected is at least $c_1 \log n$ nodes away from both v and the next fixed node since $c_2 > 2c_1$, and is clearly at most $c_2 \log n$ away from v and from its next node. Thus Invariant 3.2 is preserved. $\square$

3.3 Performance

The load balancing performace follows readily from the invariants:

Theorem 3.5 *Every node's range is of size $\Theta(\frac{1}{n})$.*

Proof. We can apply Lemma 2.4 in conjunction with Invariant 3.2 to show that the total range between any two adjacent fixed nodes is $\Theta(\frac{\log n}{n})$. Invariant 3.3 ensures that this range is divided evenly, so that every individual node's share is $\Theta(\frac{1}{n})$. $\square$

3.4 Cost

Since edges are designed to go to the nearest node succeeding a point on the circle, they may need to change as nodes are moved to correctly go to the point's successor. Only edges with one or both endpoints currently in the set being remapped can be affected. Since nodes are regularly spaced throughout the circle, if the size of the set of ideal relative edges used for construction is d the total number of edges changed is at most

$O(d \cdot \log n)$. The cost of token lookup is unchanged. The greatest cost for this algorithm is in the number of token movements, which we analyze below.

Suppose that we begin with a neighborhood of $2k$ nodes, and add a new node. Suppose for the sake of simplicity that prior to insertion the existing $2k$ nodes occupied a uniform distribution of the range, and that the new node will receive the middle portion of this range. Each of the $2k$ nodes loses a piece of size $\frac{1}{2k(2k+1)}$ (of the total range held by the $2k$ nodes) due to compression (Figure 4). Each range must shift towards the outside to fill the missing pieces and shifts of other nodes. Consider only the left half, since the right is symmetrical, and suppose the nodes are labelled 1 through k. Node 1 does not shift. Node 2 shifts the total amount of node 1's loss, $\frac{1}{2k(2k+1)}$. In general, node i must shift to compensate for the total loss of nodes 1 through $i-1$, or $\sum_{j=1}^{i-1} \frac{1}{2k(2k+1)}$; node i also loses a piece of its range. The total fraction of the range moved, therefore, is:

$$2 \cdot \sum_{i=1}^{k} \sum_{j=1}^{i} \frac{1}{2k(2k+1)} > \frac{1}{4}.$$

Combining this with Lemma 2.4, with $k = O(\log n)$, each node insertion is expected to require at least a $O(\frac{\log n}{n})$ fraction of the total number of tokens in the system to be moved. Node deletions are similar.

3.5 Degree and congestion balancing

Point remapping allows the structure to balance in more than just load distribution. We assume that edges in the structure are determined by a set of d relative ideal points on the circle (for example halfway around the circle and a quarter of the way around), and that each node connects to the nearest node on the circle after each relative ideal point.

Theorem 3.6 *All nodes have degree $O(d)$.*

Proof. Clearly each node has at most $d+2$ edges out to other nodes, at most d to the relative ideal point successors and 2 to the left and right neighbors. Now consider an arbitrary node v, which we know has a corresponding $\Theta(\frac{1}{n})$-sized range assigned to it. There is a region of this same size for each relative ideal point, backwards from v's assigned range, and any node whose point is within this range will have an edge to v. Since each node must have a $\Theta(\frac{1}{n})$-sized range for itself, at most $O(1)$ nodes can be in each of these ranges. Then each node has $O(d)$ incoming neighbors, for total degree $O(d)$. $\square$

We consider the random routing problem, where each node has a packet to send to a random destination, and we consider a structure which uses hypercubic edges (so that the ideal virtual points are $\frac{1}{2^i}$ for i from 1 to $\log n$). We assume that the destination is some token being searched for, or a point selected uniformly at random in the $[0, 1)$ circle. Consider the ideal virtual path from a node to its destination (where every successor of the ideal point followed is located at the ideal point); consider all ideal virtual paths from all nodes to their destinations together.

Lemma 3.7 *The expected number of virtual paths passing through an arbitrary $\frac{1}{n}$ sized subinterval A of the $[0, 1)$ circle is $O(\log n)$.*

Proof. See appendix. $\square$

We propose a correction to the routing process from the node to its destination to more closely follow the virtual path. As the packet is forwarded, its ideal virtual path $\{p_0, p_1, \ldots\}$ is passed with it (or computed on-the-fly); the ideal virtual path is a set of points on the circle starting with the source where successive points differ by decreasing ideal virtual points. At any intermediate node on the taken path, if one of its predecessors is closer to the preceding point on the ideal virtual path, the packet is sent back to that node; this process is repeated until the closest successor to the point is found, and then the next edge is taken. Through this we maintain the following:

Lemma 3.8 *Every node on the actual path from a source to its destination is within $O(\frac{1}{n})$ of a point on the corresponding ideal virtual path.*

Proof. See appendix. □

Theorem 3.9 *Congestion for the random routing problem in a structure with hypercubic edges where points are assigned to nodes using using point remapping is $O(\log n)$ in expectation.*

Proof. The maximum expected congestion in an interval of size $\frac{1}{n}$ of ideal virtual paths according to Lemma 3.7 is $O(\log n)$; since each node has a range of size $\Theta(\frac{1}{n})$, at most a constant number of these intervals can cause congestion in a node. According to the proof of Lemma 3.8, each real path requires $O(1)$ local hops for each point on the ideal virtual path, and consequently can cause at most $O(1)$ times the congestion of the ideal path. The total congestion on any node, then, is $O(\log n)$. □

4 Range sharing

4.1 Algorithm

In the range sharing algorithm, we map each node to a range of size $\frac{c \log n}{n}$ for a fixed constant c within the circular $[0, 1)$ interval centered around the point assigned to the node by the original DHT; these ranges will overlap, since the total size of all the ranges is $c \log n$. Each token is mapped to a point at random as before, which may lie in several ranges; we assume c is sufficiently large for the token to lie in some range. An intuitive approach is to map the token uniformly at random to the set of ranges which contain the point. While this is effective for load balancing, it is somewhat expensive to implement in a distributed system, since some node must be selected to perform this random selection, and the set of nodes which overlap each point in each node's range must be stored separately so that this selection can be performed. Instead, we store at each node v the set of all other nodes $G(v)$ which have ranges which in any amount overlap v's range; see Figure 5. We use a local hash table stored at v to randomly map to $v \cup G(v)$ all tokens which in the original DHT would be assigned to v. Note that $G(v)$ represents all nodes which v will assign tokens to in its local hash table, and also all nodes which will assign v tokens in their local hash tables. Where the intuitive algorithm would require each node to maintain multiple local hash tables, our algorithm requires only one.

To make this practical under dynamic network conditions, we use a local consistent hashing structure. Each node v constructs a local hash table for itself and the nodes in $G(v)$ by mapping each node to $O(\log n)$ virtual points in the circle and assigning each token to a random point in the circle. When a node u is assigned a token by node v, u keeps the token (it does not map the token to its own hash table).

This algorithm is similar to the SHARE algorithm[2] for load balancing in storage systems of nonuniform nodes. SHARE assigns ranges of nonuniform size to each node proportional to the node's desired capacity with overlap, and maps tokens uniformly at random to the ranges which contain the token's assigned point. Our strategy is different in that each token mapped to v initially is mapped to all of $v \cup G(v)$, including nodes which do not contain the token's point. This modification keeps each node from having to maintain more than one local consistent hashing structure.

4.2 Cost

Each node v must keep an edge to each node in $G(v)$ to be able to send tokens directly to it. Since the range is of size $\frac{c \log n}{n}$, all nodes in $G(v)$ will have starting points in a range of size $\frac{2c \log n}{n}$. Lemma 2.4 shows that the number of nodes in $G(v)$, and thus the node's additional degree, is $O(\log n)$ with high probability; assuming c is sufficiently large, Chernoff's bound can be used to show $G(v)$ has $\Omega(\log n)$ nodes with high probability. The node must also keep $O(|G(v)|^2) = O(\log^2 n)$ local storage. Token lookup is efficient since at most one additional node must be contacted.

When some node w leaves the system, each node u in $G(w)$ (nodes for which $w \in G(u)$) will need to update its consistent hash table, and as in original DHT's the tokens currently handled by w's consistent hash table will all need to be moved to the table of the new node responsible for them. Since $|G(w)|$ is $\Theta(\log n)$ with high probability the cost of processing a leave operation is $\Theta(\log n)$ plus the cost of token movements. The same holds for a join.

One optimization reduces token movement cost: force all nodes to use a single function in their local hash tables to generate random virtual points. Then the only tokens that need to be moved for leave and join operations are tokens held by the departing and the joining node respectively.

4.3 Performance

The following lemma follows from the original consistent hashing analysis using virtual nodes:

Lemma 4.1 *For each node w such that $v \in G(w)$, if $l(w)$ tokens are hashed to w using the original DHT, v will receive $\Theta(\frac{l(w)}{|G(w)|})$ tokens from w's local hash table.*

Using this we can show:

Theorem 4.2 *Each node in the range sharing algorithm receives a $O(\frac{1}{n})$ share of the tokens in the system with high probability.*

Proof. With Lemma 4.1 and the fact that $|G(v)| = \Theta(\log n)\ \forall\ v$ with high probability, the load on v is $\sum_{w \in G(v)} \Theta(\frac{l(w)}{|G(w)|}) = \sum_{w \in G(v)} \Theta(\frac{l(w)}{\log n})$. By Lemma 2.4, since $|G(v)|$ is $\Theta(\log n)$, if the total number of tokens in the system is T, we can say $\sum_{w \in G(v)} l(w) = O(\frac{T \cdot \log n}{n})$, and thus the total load on v is $O(\frac{T}{n})$. $\square$

4.4 Degree and congestion balancing

4.4.1 Edge connections

Since the points are left at their original locations, additional work must be invested to rebalance the edges. We map edge endpoints using a similar process to token placement. As before, suppose that in the original system edges are assigned based on a set of relative ideal points P with $|P| = d$ by connecting a node v to the successor of each ideal point $p \in P$. Instead of using the successor w of each p, we find the set of nodes $G(w)$ with ranges which overlap with the range assigned to w, and connect v uniformly at random to a node in $G(w) \cup w$ using w's local consistent hash table.

4.4.2 Degrees

The out-degree is $O(d + \log n)$ (d for the relative ideal point edges and $O(\log n)$ for the neighborhood). Without any improvements, selecting a random hash function to map any set of nodes to the $[0, 1)$ circle will give some node a region of size $\Omega(\frac{\log n}{n})$ with high probability, or with high probability there is a node with in-degree $\Omega(d \cdot \log n)$. Our edge connections improve this:

Theorem 4.3 *The in-degree of every node is $O(d + \log n)$ with high probability.*

Proof. See appendix. □

4.4.3 Routing

We make one modified assumption to allow a simpler routing proof: we assume that the neighborhood maintained by each node contains one additional node at each end, the node beyond the furthest node with an overlapping range, which is never assigned tokens or edges using range sharing. Since this node is not used in any hash tables, it is easy to maintain these connections across node joins and leaves. The modifications to the routing process itself resemble those used in point remapping: the new edges can cause the endpoints to vary from the points in the ideal virtual path, so corrections are used to keep these low. But in this structure, points are not moved, so the deviation from the virtual path can be much larger. We also must adjust the routing process so that paths are not collected at nodes of large ranges.

Instead of adjusting to reach the successor of the next point on the ideal virtual path, we adjust to reach *any node which has the next point on the ideal virtual path within its neighborhood*. Suppose that the ideal virtual path is a set of points $(p_0, p_1, \ldots)$ where p_0 is the start.

Lemma 4.4 *At every step of the routing process, the lookup request is either at a node which has some node of the ideal virtual path within its neighborhood, or is at a node which has such a node in its neighborhood.*

Proof. See appendix. □

At the tail end of this process, the node succeeding the token can be contacted (since it is in the neighborhood of any node whose range includes the token's point) to compute the node holding the token, and can then forward the lookup request to the node holding the token. This directly implies the following:

Theorem 4.5 *Token lookup requires at most twice as many hops and messages in the degree-balanced structure as in the original.*

4.4.4 Congestion

Again consider the random routing problem on a hypercubic structure, where our ideal virtual points are $\frac{1}{2^i}$ for i from 1 to $\log n$. We define the constant c to be large enough for any subinterval of size $\frac{c \log n}{n}$ to contain $\Theta(\log n)$ nodes with high probability. For some other constant k, we can say that there are at most $k \log n$ nodes in any interval of size $\frac{c \log n}{n}$ with high probability.

Lemma 4.6 *The expected number of virtual paths passing through an arbitrary $\frac{c \log n}{n}$ sized subinterval B of the $[0, 1)$ circle is $O(\log^2 n)$.*

Proof. See appendix. □

Theorem 4.7 *Congestion for the random routing problem on a structure with hypercubic edges mapped according to range sharing is $O(\log n)$ in expectation.*

Proof. Since every edge going into a neighborhood is randomly mapped to some nearby node, each node is the direct endpoint of an edge used by one of the $O(\log^2 n)$ virtual paths passing through its neighborhood with probability $\Theta(\frac{1}{\log n})$, or the number of paths passing reaching a node not through correction is $O(\log n)$. A node v may also receive packets as corrections from nodes outside the neighborhood of the nearby ideal virtual points, but there are only $\Theta(1)$ nodes sending corrections to v in expectation, so the expected congestion is still $O(\log n)$. □

References

[1] M. Adler, E. Halperin, R. Karp, and V. Vazirani. A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proceedings of the 35th ACM Symposium on the Theory of Computing*, 2003.

[2] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement strategies for non-uniform capacities. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 53–62, 2002.

[3] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Second International Workshop on Peer-to-Peer Systems*, 2003.

[4] B. Ghosh, F. T. Leighton, B. M. Maggs, S. M. nan, C. G. Plaxton, R. Rajaraman, A. W. Richa, R. E. T. n, and D. Zuckerman. Tight analyses of two local load balancing algorithms. *SIAM Journal on Computing*, 29(1):29–64, 2000.

[5] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 41–52, Aug. 2002.

[6] F. Kaashoek and D. Karger. Koorde: A simple degree-optimal distributed hash table. In *Second International Workshop on Peer-to-Peer Systems*, 2003.

[7] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed cashing protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, El Paso, TX, May 4–6, 1997. ACM Press, New York, NY.

[8] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2003.

[9] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[10] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Second International Workshop on Peer-to-Peer Systems*, 2003.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 161–172. ACM Press, 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, 2001.

[13] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[14] C. Xu and F. Lau. *Load Balancing in Parallel Computers, Theory and Practice*. Kluwer Academic Publishers, 1997.

Appendix 1: Figures

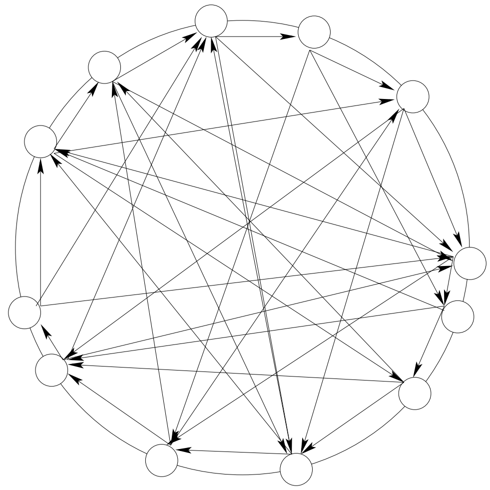

Figure 1: A sample consistent hashing structure.

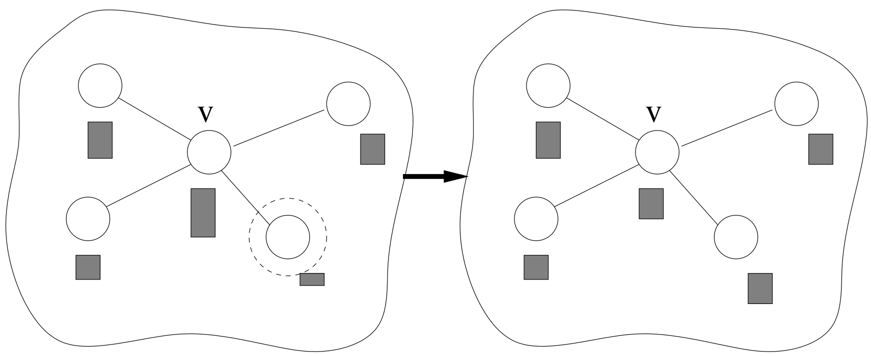

Figure 2: A node v balancing with one of its neighbors.

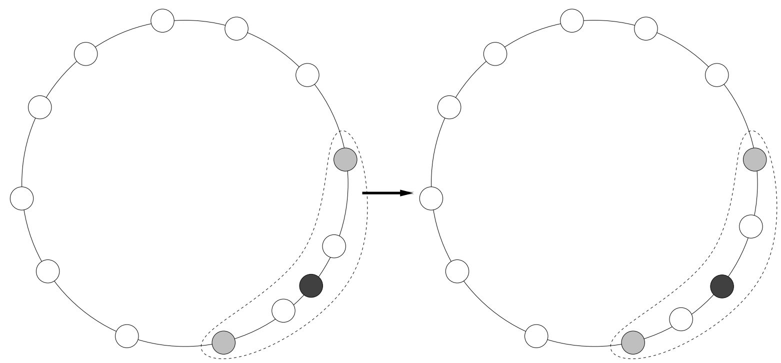

Figure 3: Inserting a node and rebalancing.

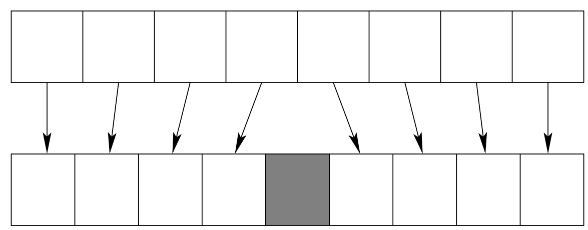

Figure 4: The shift of ranges after a node insertion.

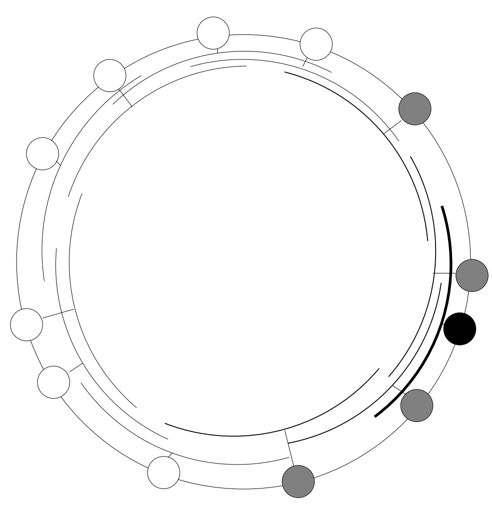

Figure 5: A set of nodes and ranges; one node v and its set $G(v)$.

Appendix 2: Additional Proofs

Proofs for Section 3

Proof of Lemma 3.7: Consider an arbitrary single interval A. For each i, there is an interval A_i which is located $\frac{1}{2^i}$ further back on the circle, so that any ideal path taking edge i from A_i will reach A. Clearly the packets originating and terminating in A will pass through A. For some constant c, we can say that there are at most c nodes in any interval of size $\frac{1}{n}$ by Theorem 3.5; then at most c packets will originate in A. Clearly we expect only one packet to terminate in A.

Consider $i = 1$, and the packets that reach A by a hop of size $\frac{1}{2^i}$. There are at most c nodes located in A_1, each of which can only initiate one packet. Since the node's destinations are selected uniformly at random, we expect half of these packets to take the $\frac{1}{2}$ edge; we expect $\frac{c}{2}$ of the packets from A_1 to advance to A, and we expect $\frac{c}{2}$ of the packets originating in A to stay in A. The total number of packets passing through A (considering start, finish, and edge 1) is $c + 1 + \frac{c}{2}$.

Now consider arbitrary i, and suppose by an inductive hypothesis that the current number of packets in every interval is at most c, including the interval A_i. The endpoints of these packets are randomly determined, but since all edges through $i - 1$ have been considered, half of the endpoints in A_i will be beyond A (causing edge i to be taken), so we expect $\frac{c}{2}$ of the packets in A_i to go to A.

The total number of packets passing through A, then, is $c + 1 + (\log n \cdot \frac{c}{2})$, which is $O(\log n)$. □

Proof of Lemma 3.8: The source v_0 is the point p_0 on the ideal virtual path. After the first edge is taken to some node v_1, v_1 is by definition the closest successor to p_1, since otherwise the source's edges would be inaccurate. Since each node's range is of size $O(\frac{1}{n})$, v_1 must be within $O(\frac{1}{n})$ of p_1; let us say that this is at most $\frac{c}{n}$ for some constant c. The next edge taken will go to v_2, the successor of an edge corresponding to the same ideal relative point as taken in the ideal virtual path; the ideal endpoint of this edge will vary from the next point on the ideal virtual path by the same amount v_1 varies from p_1, which is $\frac{c}{n}$. Then the actual successor v_2 will vary from this point by up to $\frac{c}{n}$, which is up to $\frac{2c}{n}$ away from p_2. But some node must be within $\frac{c}{n}$ of p_2, so v_2 will send the request back until this node is found. Furthermore, since each node has a range of $\Theta(\frac{1}{n})$, there can only be a constant number of nodes in between v_2 and p_2. This process continues until the destination is reached. □

Proofs for Section 4

Proof of Theorem 4.3: We consider in-degree based on relative ideal point edges first. Any node with a relative ideal point in the range of a node v or in the range of a node in v's neighborhood of $O(\log n)$ nodes has a chance to connect to v. Lemma 2.4 and Chernoff's bound show that there are $\Theta(\log n)$ such nodes for each ideal point, or $\Theta(d \cdot \log n)$ nodes total. If all neighborhoods are the same size, each node connects to v with the same probability, $O(\frac{1}{\log n})$, and the expected in-degree of each node is $O(d)$. Chernoff's bounds can be used to show that every node's in-degree is $O(d + \log n)$ with high probability. $O(\log n)$ additional incoming edges come from neighbors, but the total in-degree is still $O(d + \log n)$. □

Proof of Lemma 4.4: Proof is by induction. The first step in the routing process takes the edge corresponding to some relative ideal point p; the destination of this edge, w_1, is decided by the successor of this point v_1. Edge endpoints are assigned within the neighborhood, so w_1 is in v_1's neighborhood; neighborhoods are symmetric, so v_1 is in w_1's as well. Since w_1 is in v_1's neighborhood, and since neighbors extend one node beyond the assignment of edges, w_1's neighborhood contains all of the range in which points are mapped to v_1, including the point p_1. Routing can proceed from w_1 without correction.

Now assume we are at some node w_i which has point p_i in its neighborhood, which implies that the distance between w_i and p_i is at most $\frac{c \log n}{n}$ for some constant c. We take the edge from w_i corresponding to the difference between p_{i+1} and p_i to reach some node q. If node q has p_{i+1} in its neighborhood, it becomes w_{i+1}; this is not necessarily true. But node q was assigned the edge from w_i by the node x which was the successor of the point $w_i + (p_{i+1} - p_i)$, which must be within $\frac{c \log n}{n}$ of point p_{i+1}. The node x itself may not be close enough, but its neighbor towards p_{i+1} must be, and q's neighborhood must extend to this node past x by definition. Therefore some node in q's neighborhood can become w_{i+1} and the inductive hypothesis holds. □

Proof of Lemma 4.6: The proof is comparable to the proof of Lemma 3.7, since $k \log n$ packets are expected to be in every subinterval B_i, and half of each will pass through B. □