Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems

David R. Karger
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
karger@csail.mit.edu

Matthias Ruhl
IBM Almaden Research Center
San Jose, CA 95120, USA
ruhl@almaden.ibm.com

ABSTRACT

Load balancing is a critical issue for the efficient operation of peer-to-peer networks. We give two new load-balancing protocols whose provable performance guarantees are within a constant factor of optimal. Our protocols refine the *consistent hashing* data structure that underlies the Chord (and Koorde) P2P network. Both preserve Chord's logarithmic query time and near-optimal data migration cost.

Consistent hashing is an instance of the distributed hash table (DHT) paradigm for assigning items to nodes in a peer-to-peer system: items and nodes are mapped to a common address space, and nodes have to store all items residing closeby in the address space.

Our first protocol balances the distribution of *the key address space* to nodes, which yields a load-balanced system when the DHT maps items "randomly" into the address space. To our knowledge, this yields the first P2P scheme simultaneously achieving $O(\log n)$ degree, $O(\log n)$ look-up cost, and constant-factor load balance (previous schemes settled for any two of the three).

Our second protocol aims to directly balance the distribution of *items* among the nodes. This is useful when the distribution of items in the address space cannot be randomized. We give a simple protocol that balances load by moving nodes to arbitrary locations "where they are needed." As an application, we use the last protocol to give an optimal implementation of a distributed data structure for range searches on ordered data.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications—*load balancing*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

Peer-to-peer systems, load balancing

SPAA'04, June 27–30, 2004, Barcelona, Spain.

1. INTRODUCTION

Peer-to-peer (P2P) systems are a current research focus in computer systems and networking. Such systems are attractive in their potential to harness the vast distributed computation and storage resources in today's networks, without need for complex and sensitive centralized control.

A core problem in peer-to-peer systems is the distribution of items to be stored or computations to be carried out to the nodes that make up the system. A particular paradigm for such allocation, known as the *distributed hash table (DHT)*, has become the standard approach to this problem in research on peer-to-peer systems [7, 9, 12, 13, 16, 17, 21]. A distributed hash table interface implements a hash function that maps any given item to a particular machine ("bucket") in the peer-to-peer network. For example, Chord [21] uses Consistent Hashing [8] to assign items to nodes: items and nodes are pseudo-randomly hashed to a circular address space (represented by the interval $[0,1]$ with addresses 0 and 1 identified), and a node stores all items whose addresses fall between the node's own address and the address of the node preceeding it in the address space.

DHTs differ from traditional hash tables in two key ways: First, in addition to the insertion and deletion of items, DHTs must support the insertion and deletion of *buckets*: as machines join and leave the network, items must be migrated to other machines and the hash function revised to reflect their new location. Second, some kind of *routing protocol* is usually necessary: since it is not feasible in a P2P system for every node to maintain up-to-date knowledge of all other nodes in the system, an item is looked up (or inserted) by following a sequence of routing hops through the peer-to-peer network.

A large number of algorithms have been proposed (and implemented in systems) [16, 21, 9, 17, 13, 7] to provide distributed hash table functionality. The majority of them require each node of an n-node P2P system to keep track of only $O(\log n)$ "neighbor nodes" and allow the machine responsible for any key to be looked up and contacted in $O(\log n)$ routing hops. Recently some variants have been proposed [12, 7] that support $O(\log n)$-hop lookups with only constant neighbor degree; this is theoretically optimal but may be undesirable in practice in light of fault-tolerance considerations.

1.1 Load Balancing

An important issue in DHTs is load-balance—the even distribution of items (or other load measures) to nodes in the DHT. All DHTs make some effort to load-balance, generally by (i) randomizing the DHT address associated with each item with a "good enough" hash function and (ii) making each DHT node responsible for a balanced portion of the DHT address space. Chord is a prototypical example of this approach: its "random" hashing of nodes to a ring means that each node is responsible for only a small interval of ring address space, while the random mapping of items means that only a limited number of items land in the (small) ring interval owned by any node.

This attempt to load-balance can fail in two ways. First, the typical "random" partition of the address space to nodes is not completely balanced. Some nodes end up responsible for a larger portion of the addresses and thus receive a larger portion of the randomly distributed items. Second, some applications may preclude the randomization of data items' addresses. For example, to support range searching in a database application the items may need to be placed in a specific order, or even at specific addresses, on the ring. In such cases, we may find the items unevenly distributed in address space, meaning that balancing the address space among nodes does not balance the distribution of items among nodes. We give protocols to solve both of the load balancing challenges just described.

1.1.1 Address-Space Balancing

In general, distributed hash tables do not offer load balance quite as good as standard hash tables. A typical standard hash table partitions the space of possible hash-function values evenly over the buckets; thus, assuming the hash function is "random enough" and sufficiently many keys are inserted, those keys will be evenly distributed among the buckets. Current distributed hash tables do *not* evenly partition the address space into which keys get mapped; some machines get a larger portion of it. Thus, even if keys are numerous and random, some machines receive more than their fair share, by as much as a factor of $O(\log n)$ times the average.

To cope with this problem, most DHTs use *virtual nodes*: each real machine pretends to be several distinct machines, each participating independently in the DHT protocol. The machine's load is thus determined by summing over several virtual nodes', creating a tight concentration of (total) load near the average. As an example, the Chord DHT is based upon consistent hashing [8], which requires $O(\log n)$ virtual copies to be operated for every node.

Virtual nodes have drawbacks. Most obviously, the real machine must allocate space for the data structures of each virtual node; more virtual nodes mean more data structure space. However, P2P data structures are typically not that space-expensive (requiring only logarithmic space per node) so multiplying that space requirement by a logarithmic factor is not particularly problematic. A much more significant problem arises from network bandwidth. In general, to maintain connectivity of the network, every (virtual) node must frequently ping its neighbors, make sure they are still alive, and replace them with new neighbors if not. Running multiple virtual nodes creates a multiplicative increase in the (very valuable) network bandwidth consumed for maintenance.

Below, we will solve this problem by arranging for each node to activate *only one* of its $O(\log n)$ virtual nodes at any given time. The node will occasionally check its inactive virtual nodes, and may migrate to one of them if the distribution of load in the system has changed. Since only one virtual node is active, the real node need not pay the original Chord protocol's multiplicative increase in space and bandwidth costs. As in the original Chord protocol, our scheme gives each real node only a small number of "legitimate" addresses on the Chord ring, preserving Chord's (limited) protection against address spoofing by malicious nodes trying to disrupt the routing layer. (If each node could choose an arbitrary address, then a malicious node aiming to erase a certain item could take responsibility for that item's key and then refuse to serve the item.)

1.1.2 Item Balancing

A second load-balancing problem arises from certain database applications. A hash table randomizes the order of keys. This is problematic in domains for which order matters—for example, if one wishes to perform range searches over the data. This is one of the reasons binary trees are useful despite the faster lookup performance of hash tables. An order-preserving dictionary structure cannot apply a randomized (and therefore load balancing) hash function to its keys; it must take them as they are. Thus, even if the address space is evenly distributed among the nodes, an uneven distribution of the keys (e.g., all keys near 0) may lead to all load being placed on one machine.

In our work, we develop a load balancing solution for this problem. Unfortunately, the "limited assignments" approach discussed for key-space load balancing does not work in this case—it is easy to prove that if nodes can only choose from a few addresses, then certain load balancing tasks are beyond them. Our solution to this problem therefore allows nodes to move to arbitrary addresses; with this freedom we show that we can load-balance an arbitrary distribution of items, without expending much cost in maintaining the load balance.

Our scheme works through a kind of "work stealing" in which underloaded nodes migrate to portions of the address space occupied by too many items. The protocol is simple and practical, with all the complexity in its performance analysis.

Our protocol can also be used to balance weighted items, where the weight of an item can for example reflect its storage size, or its popularity and the resulting bandwidth requirements.

1.2 Our Contributions

In this paper we give two distributed load-balancing schemes for data storage applications in P2P networks.

First, in Section 2, we give a protocol that improves consistent hashing in that every node is responsible for a $O(1/n)$ fraction of the address space with high probability, without using virtual nodes. The protocol is dynamic, with an insertion or deletion causing $O(\log\log n)$ other nodes to change their positions. Each node has a fixed set of $O(\log n)$ possible positions that it chooses from. This set only depends on the node itself (computed e.g. as hashes of the node IP address), impeding malicious spoofing attacks on the network. Another nice property of this protocol is that the "appropriate" state of the system (i.e., which virtual nodes are active), although random, is *independent* of the history of item and node arrivals and departures. This Markovian property means that the system can be analyzed as if it were static, with a fixed set of nodes and items; such analysis is generally much simpler than a dynamic, history-dependent analysis.

Combining our load-balancing scheme with the Koorde routing protocol [7], we obtain a protocol that simultaneously offers (i) $O(\log n)$ degree per real node, (ii) $O(\log n/\log\log n)$ lookup hops, and (iii) constant factor load balance. Previous protocols could achieve any two of these but not all three—generally speaking, achieving property (iii) required operating $O(\log n)$ virtual nodes, which pushed the degree to $O(\log^2 n)$ and therefore failed to achieve property (i).

A second interpretation of our results can be given independent of P2P systems. Consistent hash functions [8] are useful generalized hash functions assigning items to buckets in a dynamic fashion that allows both items and buckets to be inserted and deleted dynamically. The initial implementation of consistent hashing, however, required $O(n\log n+N)$ space to store N items in n buckets. Our new scheme reduces the necessary space allocation to the optimal $O(n+N)$ space, at the cost of slower bucket insertions and deletions (the insertion or deletion of a node causes $O(\log\log n)$ buckets to change in expectation, compared to $O(1)$ buckets in the worst case for the original version of consistent hashing). It is an interesting open problem to optimize space *without* affecting bucket insertion and deletion time.

In the second part of our work, we consider arbitrary distributions of keys, which forces us to allow nodes to move to arbitrary addresses. In Section 3, we give a dynamic protocol that changes nodes' addresses in order to achieve load balance. The protocol is randomized, and relies on the underlying P2P routing framework to be able to contact "random" nodes in the system. We show that the amortized rebalancing costs in terms of number of items moved are $O(N/n)$ for a node insertion or deletion (where N is the number of items in the system), and $O(1)$ for the insertion or deletion of an item. The protocol does not require the knowledge of N or n for operation, and it can be extended to items with different weights or storage costs.

In particular, this load balancing protocol can be used to store ordered data, such that the items are not hashed, but mapped to the DHT address space in an order-preserving way. Every node then stores the items falling into a continuous segment of that ordering. In Section 3.2.4, we describe how this can be used to implement a range-search data structure, where given items a and b, the data structure is to return all items x stored in the system that satisfy $a \leq x \leq b$. We give the first such protocol that achieves an $O(\log n + Kn/N)$ query time (where K is the size of the output).

We design our solutions in the context of the Chord (and Koorde) DHT [21] but our ideas seem applicable to a broader range of DHT solutions. Chord [21] uses Consistent Hashing to assign items to nodes, achieving key-space load balance using $O(\log n)$ virtual nodes per real node. On top of Consistent Hashing, Chord layers a routing protocol in which each node maintains a set of $O(\log n)$ carefully chosen "neighbors" that it uses to route lookups in $O(\log n)$ hops. Our modifications of Chord are essentially modifications of the Consistent Hashing protocol assigning items to nodes; we can inherit unchanged Chord's neighbor structure and routing protocol. Thus, for the remainder of this paper, we ignore issues of routing and focus on the assignment problem.

1.3 Related Work

While much research has been done on routing in P2P networks, work on efficient load balancing and complex queries in P2P is only in its beginning stages. Most structured P2P systems simply assume that items are uniformly distributed on nodes.

Two protocols that achieve near-optimal load-balancing without the use of virtual nodes have recently been given [1, 14]. Our scheme improves upon them in three respects. First, in those protocols the address assigned to a node depends on the rest of the network, i.e. the address is *not* selected from a list of possible addresses that only depend on the node itself. This makes the protocols more vulnerable to malicious attacks. Second, in those protocols the address assignments depend on the construction history, making them harder to analyze. Third, their load-balancing guarantees are only shown for the "insertions only" case, while we also handle deletions of nodes and items.

Work on load balancing by moving items can be found in work of Rao et al. [15]. Their algorithm is similar to ours, however it only works when the set of nodes and items are fixed (i.e. without insertions or deletions), and they give no provable performance guarantees, only experimental evaluations.

A theoretical analysis of a similar protocol was given by Anagnostopoulos, Kirsch and Upfal [2]. In their setting, however, items are assumed to be jobs that are executed at a fixed rate, i.e. items disappear from nodes at a fixed rate. Moreover, they analyze the average wait time for jobs, while we are more interested in the total number of items moved to achieve load balance.

Complex queries such as range searches are also an emerging research topic for P2P systems [5, 6]. An efficient range search data structure was recently given by Aspnes and Shah [3]. However, that work does not address the issue of load balancing the number of items per node, making the simplifying assumption that each node stores only one item. In this setting, the lookup times are $O(\log N)$ in terms of the number of items N, and not in terms of the number of nodes n. Also, $O(\log N)$ storage is used per data item, meaning a total storage of $O(N \log N)$, which is typically much worse than $O(N + n\log n)$.

In recent independent work, Ganesan and Bawa [4] consider a load balancing scheme similar to ours and point out applications to range searches. However, their scheme relies on being able to quickly find the least and most loaded nodes in the system. It is not clear how to support this operation efficiently, and without creating heavy network traffic for these nodes with extreme load.

1.4 Notation

In this paper, we will use the following notation.

n is the number of nodes in system
N is the number of items stored in system (usually $N \gg n$)
ℓ_i is the number of items stored at node i
$L = N/n$ is the average (desired) load in the system

Whenever we talk about the address space of a P2P routing protocol (such as Chord), we assume that this space is normalized to the interval $[0,1]$. We further assume that the addresses 0 and 1 are identified, i.e. that the address space forms a ring.

2. ADDRESS-SPACE BALANCING

In this section we give a protocol that improves consistent hashing in that every node is responsible for a $O(1/n)$ fraction of the address space with high probability (whp), without use of virtual nodes. This improves space and bandwidth usage in Chord by a logarithmic factor over traditional consistent hashing. The protocol is dynamic, with an insertion or deletion causing $O(\log\log n)$ other nodes to change their positions. Each node has a fixed set of $O(\log n)$ possible positions (called "potential nodes"); it chooses exactly one of those potential nodes to become *active* at any time — this is the only node that it actually operates. A node's set of potential nodes depends only on the node itself (their addresses computed e.g. as hashes $h(i,1), h(i,2), \ldots, h(i, c\log n)$ of the node identifier i), making attacks on the network more difficult.

We denote the address $(2b+1)2^{-a}$ by $\langle a,b\rangle$, where a and b are integers satisfying $0 \leq a$ and $0 \leq b < 2^{a-1}$. This yields an unambiguous notation for all addresses with finite binary representation. We impose an ordering $\prec$ on these addresses according to the *length* of their binary representation (breaking ties by magnitude of the address). More formally, we set $\langle a,b\rangle \prec \langle a',b'\rangle$ iff $a < a'$ or ($a = a'$ and $b < b'$). This yields the following ordering:

$$0 = 1 \prec \frac{1}{2} \prec \frac{1}{4} \prec \frac{3}{4} \prec \frac{1}{8} \prec \frac{3}{8} \prec \frac{5}{8} \prec \frac{7}{8} \prec \frac{1}{16} \prec \ldots$$

We are now going to describe our protocol in terms of the "ideal" state it wants to achieve.

Ideal state: Given any set of active nodes, each (possibly inactive) potential node "spans" a certain range of addresses between itself and the succeeding active node on the address ring. Each real node has activated the potential node that spans the minimal (under the ordering $\prec$) address.

Our protocol consists of the simple update rule that any node for which the ideal state condition is not satisfied, instead activates the potential node for which the condition is (locally) satisfied. In other words, each node occasionally determines which of its $O(\log n)$ potential nodes spans the smallest address (according to $\prec$). The node then activates that particular potential node.

We will now prove that such a system has a unique ideal state. This shows that the protocol is Markovian, i.e. that the resulting state does not depend on the construction history, and there is no bias introduced by a sequence of insertions and deletions. This is similar to treaps [20] where items are inserted with random priorities, yet the result does not depend on the order of the insertions.

Theorem 1 *The following statements are true for the above protocol, if every node has $c\log n$ potential addresses that are chosen $\Omega(\log n)$-independently at random.*

(i) For any set of nodes there is a unique ideal state.

(ii) Given any starting state, the local improvements will eventually lead to this ideal state.

(iii) In the ideal state of a network of n nodes, whp all neighboring pairs of active nodes will be at most $(4+\varepsilon)/n$ apart, for any $\varepsilon \leq 1/2$ with $c \geq 1/\varepsilon^2$. (This bound improves to $(2+\varepsilon)/n$ for very small ε.)

(iv) Upon inserting or deleting a node into an ideal state, in expectation at most $O(\log\log n)$ nodes have to change their addresses for the system to again reach the ideal state.

2.1 Proof of Theorem 1

The unique ideal state can be constructed as follows. The potential node immediately preceding address 1 will be active, since its real-node owner has no better choice and cannot be blocked by any other node from spanning address 1. That real node's other potential nodes will then be out of the running for activation. Of the remaining potential nodes, the one closest to address 1/2 will become active for the same reason, and so on. We continue in this way down the $\prec$-ordered list of addresses. This greedy process clearly defines the unique ideal state, showing claim (i).

We will show claim (ii) by arguing that every local improvement reduces the distance from the current state to the ideal state (with "distance" appropriately defined). For this, let A be the set of addresses with finite binary expansion, i.e. the addresses in the ordering $\prec$. Fix the set X of active nodes in the system. We define a function $f_X : A \to (\mathbb{R} \cup \infty)$ as follows. For each address $a \in A$, consider the active node x that spans the address interval containing a. If that interval does not contain any smaller (in the sense of $\prec$) address than a, then let $f_X(a)$ be the address distance between x and a, otherwise let $f_X(a) := \infty$.

Two different sets X and Y of active nodes will lead to different functions f_X and f_Y. Consider the lexicographic ordering on all of these functions, i.e. functions are compared by first considering their relative value at 0, then at $\frac{1}{2}$, $\frac{1}{4}$, $\frac{3}{4}$, and so on in the order of $\prec$, until the first unequal value determines the relative ordering of the functions. It is then straightforward to show (cf. [18, Lemma 4.5]) that

(a) among all choices of active nodes, the ideal state leads to the smallest possible function f_X under this lexicographic ordering, and

(b) every local improvement makes the function f_X become smaller (i.e. reduces the distance to the ideal state).

Combined with the facts that there is only one ideal state, i.e. only one state in which no local improvement is possible (claim (i)), and that there is only a finite number of potential-node choices for active nodes, this shows claim (ii).

To prove the remaining two claims, we will assume for simplicity that the potential nodes' addresses are chosen independently at random. But just as with the original consistent hashing scheme [10], our results continue to hold for $\Omega(\log n)$-wise independent choices of potential addresses. This follows by a standard application of results of Chernoff [19]. In our proof, we will use a Chernoff bound to show that each address interval of them form $[a-\varepsilon/n, a]$ will whp contain at least one potential node. This can be rephrased as a standard balls-and-bins experiment, to which the results of Chernoff [19] directly apply.

To prove claim (iii), recall how we constructed the ideal state for claim (i) above by successively assigning nodes to increasing addresses in the order $\prec$. In this process, suppose we are considering one of the first $(1-\varepsilon)n$ addresses in $\prec$. Consider the interval I of length ε/n preceding this address in the address space. Since this is one of the first $(1-\varepsilon)n$ first addresses, at least εn of the real nodes have not yet been given a place on the ring. Among the $c\varepsilon n\log n$ potential positions of these nodes, with high probability one will land in the length-ε/n interval I under consideration. So whp, for each of the first $(1-\varepsilon)n$ addresses in the order $\prec$, the potential node spanning that address will land within distance ε/n preceding the address. Since these first $(1-\varepsilon)n$ addresses break up the unit circle into intervals of size at most $4/n$, claim (iii) follows. Note that for very small ε, the first $(1-\varepsilon)n$ addresses actually break up the unit circle in intervals of size $2/n$, which shows the additional claim.

For claim (iv), it suffices to consider a deletion since the system is Markovian, i.e. the deletion and addition of a given node are symmetric and cause the same number of changes. Consider what happens when a node assigned to some address $\langle a_0, b_0 \rangle$ gets deleted from the network. Then some node previously assigned to an address $\langle a_1, b_1 \rangle \succ \langle a_0, b_0 \rangle$ may get reassigned to $\langle a_0, b_0 \rangle$, causing some node previously assigned to $\langle a_2, b_2 \rangle \succ \langle a_1, b_1 \rangle$ to move to $\langle a_1, b_1 \rangle$, and so on. This results in a linear sequence of changes that, when complete, have produced the ideal state. Since nodes only move to smaller (in terms of $\prec$) addresses, the number of movements is clearly finite. We will now show that this number is $O(\log\log n)$ in expectation.

Let n_i be the number of nodes assigned to active addresses $\langle a', b' \rangle$ with $\langle a', b' \rangle \succ \langle a_i, b_i \rangle$. Note that the node moving to address $\langle a_i, b_i \rangle$ is uniformly random among these n_i nodes, thus $n_{i+1} \leq n_i/2$ with probability at least $1/2$. Since this is true for all i, whp $O(\log n)$ movements suffice to achieve $\log n$ halvings, which will reduce n_i to zero. This shows that whp $O(\log n)$ nodes have to change their addresses upon the deletion or insertion of a node.

However, the exchange sequence can also stop *before* n_i is reduced to zero. This happens if none of the n_i nodes has a potential position between address $\langle a_i, b_i \rangle$ and the active node preceeding it. Since we assumed the system to be load-balanced, the distance between $\langle a_i, b_i \rangle$ and the preceeding active node is $O(1/n)$. The gap therefore contains $\ell = O(\log n)$ potential nodes whp. The probability that none of the ℓ potential nodes in the gap is owned by any of the n_i active nodes that would want to move to them is

$$\left(1 - \frac{n_i}{n}\right)^{\ell} \approx \exp(-\ell \cdot n_i/n).$$

This probability becomes a constant when $n/n_i = \Omega(\ell)$, i.e. $n_i = O(n/\log n)$, at which point we perform in expectation only a constant number of additional moves. By the above discussion, n_i halves in every step in expectation, so it takes only $\log(\frac{n}{O(n/\log n)}) =$

$O(\log\log n)$ movements in expectation to get $n_i = O(n/\log n)$. Thus the expected number of moves in total is $O(\log\log n) + O(1) = O(\log\log n)$. $\Box$

2.2 Discussion

Intuitively, the protocol achieves load-balance for two reasons. First, since the nodes prefer to be near an address that is small in the ordering $\prec$, there will be active nodes close to almost all small (according to $\prec$) addresses. Second, the addresses in any beginning segment of the ordering $\prec$ (almost) uniformly subdivide the address range $[0,1]$. Combining these two facts implies the load-balance.

We note that the above scheme is highly efficient to implement in the Chord P2P protocol, since one has direct access to the addresses of successors in the address ring. Moreover, the protocol can also function when nodes disappear without invoking a proper deletion protocol. By having every node occasionally check whether they should move, the system will eventually converge towards the ideal state. This can be done with insignificant overhead as part of the general maintenance protocols that have to run anyway to update the routing information of the Chord protocol.

One (possibly) undesirable aspect of the above scheme is that $O(\log\log n)$ nodes change their address upon the insertion or deletion of a node, because this will cause an $O(\log\log n/n)$ fraction of all items to be moved. However, since every node has only $O(\log n)$ possible positions, it can cache the items stored at previous active positions, and will eventually incur little data migration cost: when returning to a previous location, it already knows about the items stored there. Alternatively, if every real node activates $O(\log\log n)$ potential nodes instead of just 1, we can reduce the fraction of items moved to $O(1/n)$, which is optimal within a constant factor. This is shown by a straightforward variation on our analysis, using the fact that with $\log\log n$ activated nodes per real node, each move only involves on average a $1/n\log\log n$ fraction of the data. All other performance characteristics are carried over from the original scheme. It remains open to achieve $O(1/n)$ data migration *and* $O(1)$ virtual nodes while attaining all the other metrics we have achieved here.

3. ITEM BALANCING

We have shown how to balance the address space, but sometimes this is not enough. Some applications, such as those aiming to support range-searching operations, need to specify a particular, non-random mapping of items into the address space. In this section, we consider a dynamic protocol that aims to balance load for *arbitrary* item distributions. To do so, we must sacrifice the previous protocol's restriction of each node to a small number of potential node locations—instead, each node is free to migrate anywhere. This is unavoidable: if each node is limited to a bounded number of possible locations, then for any n nodes we can enumerate all the places they might possibly land, take two adjacent ones, and address all the items in between them: this assigns all the items to one unfortunate node.

Our protocol is randomized, and relies on the underlying P2P routing framework. (If the node distribution is very skewed, it might be necessary to augment the routing infrastructure, see Section 3.2.3 below.) The protocol is the following (where ε is any constant with $0 < \varepsilon < 1/4$). Recall that each node stores the items whose addresses fall between the node's address and its predecessor's address, and that ℓ_j denotes the load on node j. Here, the index j runs from $1, 2, \ldots, n$ in the order of the nodes in the address space.

Item balancing: Each node i occasionally contacts another node j at random. If $\ell_i \le \varepsilon\ell_j$ or $\ell_j \le \varepsilon\ell_i$ then the nodes perform a load balancing operation (assume wlog that $\ell_i > \ell_j$), distinguishing two cases:

Case 1: $i = j+1$**:** In this case, i is the successor of j and the two nodes handle adjacent address intervals. Node j increases its address so that the $(\ell_i - \ell_j)/2$ items with lowest addresses in i's interval get reassigned from node i to node j. Both nodes end up with load $(\ell_i + \ell_j)/2$.

Case 2: $i \neq j+1$**:** If $\ell_{j+1} > \ell_i$, then we set $i := j+1$ and go to case 1. Otherwise, node j moves between nodes $i-1$ and i to capture half of node i's items. This means that node j's items are now handled by its former successor, node $j+1$.

This protocol also quickly balances the load, starting with an arbitrary load distribution.

Lemma 2 *Starting with an arbitrary load distribution, if every node contacts $O(\log n)$ random nodes for the above protocol, then whp all nodes will end up with a load of at most $\frac{16}{\varepsilon}L$. Another round of everyone contacting $O(\log n)$ other nodes will also bring all loads to at least $\frac{\varepsilon}{16}L$.*

Proof Sketch: Since the proof is similar to the one given for Theorem 3 below, we only sketch the general outline. Consider one particular node with load at least $\frac{16}{\varepsilon}L$. If this node contacts a random node, then with probability at least $1/2$ it will be able to enter in a load exchange. Contacting $\Theta(\log n)$ other nodes will therefore lead to $\Theta(\log n)$ load exchanges whp, each reducing the node's load by a constant factor. Thus, independent of the starting load, the final load will be at most $\frac{16}{\varepsilon}L$.

For the lower bound of the load, a similar argument applies. $\Box$

To state the performance of the protocol once a load balanced state has been reached, we need the concept of a *half-life* [11], which is the time it takes for half the nodes or half the items in the system to arrive or depart.

Theorem 3 *If each node contacts $\Omega(\log n)$ other random nodes per half-life as well as whenever its own load doubles or halves, then the above protocol has the following properties.*

(i) With high probability, the load of all nodes remains between $\frac{\varepsilon}{16}L$ and $\frac{16}{\varepsilon}L$.

(ii) The amortized number of items moved due to load balancing is $O(1)$ per item insertion or deletion, and $O(L)$ per node insertion or deletion. $\Box$

3.1 Proof of Theorem 3

For part (i), consider a node with load ℓ. Suppose it enters a load exchange with a node of load $\ell' \le \varepsilon\ell$. Then the node's load reduces by a factor of at least

$$\frac{\ell}{\frac{1}{2}(\ell+\ell')} \ge \frac{\ell}{\frac{1}{2}(\ell+\varepsilon\ell)} = \frac{2}{1+\varepsilon} =: \beta.$$

Consider one particular node, whose load increases beyond $\frac{2}{\varepsilon}L$. We will show that its load will drop below $\frac{2}{\varepsilon}L$ before ever rising as high as $\frac{16}{\varepsilon}L$. For this it suffices if this node has $\log_\beta \frac{16/\varepsilon}{2/\varepsilon} = \log_\beta 8 = \Theta(1)$ successful load exchanges before its load increases to $\frac{16}{\varepsilon}L$.

By Markov's inequality, half the nodes have load at most $2L$. So if a node with load exceeding $\frac{2}{\varepsilon}L$ contacts a random node, then with probability $1/2$ it will enter a load exchange, and $\Theta(\log n)$ invocations of the protocol will lead to a load exchange whp. It therefore suffices to invoke the protocol $O(\log n)$ times before the load increases from $\frac{2}{\varepsilon}L$ to $\frac{16}{\varepsilon}L$.

What can cause the load of a node to increase? First, the number of items stored at the node can increase. Second, the value of $L = N/n$ can drop globally, either by a decrease of the number of items in the system, or an increase of the number of nodes in the system. The effective change in relative load of a node is the product of these three effects. Thus, for the relative node to change by a factor of 8 from $\frac{2}{\varepsilon}L$ to $\frac{16}{\varepsilon}L$, at least one of the values has to change by a factor of 2. So the update rate stated in the Theorem is sufficient to maintain the claimed load-balance.

A similar argument yields a lower bound on the load of all nodes, noting that the upper bound of $\frac{16}{\varepsilon}L$ on the load implies that a constant fraction of all nodes have a load of at least $L/2$. This means that a node with little load (i.e. less than $\frac{\varepsilon}{2}L$ load) is likely to contact one of these nodes at random.

For part (ii), we use a potential function argument. We show that item and node insertions and departures cause only limited increases in the potential, while our balancing operation causes a significant decrease in the potential if it is large. This potential function is

$$\Phi(\overline{\ell}) := \delta\left(\sum_{i=1}^{n} \ell_i \log \ell_i - N \log L\right),$$

where δ is a sufficiently large constant, e.g. $\delta = 8$. Recall that ℓ_i is the load of node i, i.e. the number of items stored at the node (since we are considering the unweighted case). Our potential function is related to the entropy of the item distribution. More precisely, up to an additive term independent of the item distribution, the potential function is exactly the negative of the entropy. Thus, our function gets minimized when all nodes have the same load.

The amortized cost of an operation (insertion, deletion, or load exchange) will be its actual cost plus the resulting change in potential function, i.e. "amortized cost" = "actual cost" + $\Phi_{\text{after}} - \Phi_{\text{before}}$.

Item insertion

The actual cost of inserting an item is 1, since the affected item has to be handled. So the amortized cost of inserting an item at a node j is

$$\begin{aligned}
&1+\delta\Bigg(\sum_{i\neq j}\ell_i\log\ell_i+(\ell_j+1)\log(\ell_j+1)-(N+1)\log\frac{N+1}{n}\\
&\qquad -\sum_i \ell_i\log\ell_i+N\log\frac{N}{n}\Bigg)\\
&=1+\delta\Bigg((\ell_j+1)\log(\ell_j+1)-\ell_j\log\ell_j+N\log\frac{N}{n}\\
&\qquad -(N+1)\log\frac{N+1}{n}\Bigg)\\
&=1+\delta\Bigg(\log(\ell_j+1)+\log\left(\frac{\ell_j+1}{\ell_j}\right)^{\ell_j}-\log\frac{N}{n}\\
&\qquad +\log\left(\frac{N/n}{(N+1)/n}\right)^{N+1}\Bigg)\\
&=1+\delta\Bigg(\log(\ell_j+1)+\log\left(1+\frac{1}{\ell_j}\right)^{\ell_j}-\log\frac{N}{n}\\
&\qquad +\log\left(1-\frac{1}{N+1}\right)^{N+1}\Bigg)\\
&=1+\delta\big(\log(\ell_j+1)-\log L+\Theta(1)\big)\\
&=\Theta\big(1+\log(\ell_j+1)-\log L\big).
\end{aligned}$$

If $\ell_j = O(L)$, the cost reduces to $O(1)$.

Item deletion

The actual cost of a deletion is 0, since no item has to be moved. The change in potential is the negative of an item insertion's, and thus $\Theta(1-\log \ell_j + \log L)$. This cost is $O(1)$ since $\ell_j = \Omega(L)$.

Node insertion

The actual cost of adding a new node is zero, as no items are moved. The change in potential function (and therefore the amortized cost) is

$$\begin{aligned}
-\delta N\log\frac{N}{n+1}+\delta N\log\frac{N}{n} &= \delta\log\left(\frac{n+1}{n}\right)^N = \delta\log\left(1+\frac{1}{n}\right)^{Ln}\\
&= \Theta(\log e^L) = \Theta(L).
\end{aligned}$$

Node deletion

When deleting node j and moving its items to node k, we incur an actual cost of ℓ_j items. The amortized cost of deleting a node is therefore:

$$\begin{aligned}
&\ell_j+\delta\Bigg((\ell_j+\ell_k)\log(\ell_j+\ell_k)-\ell_j\log\ell_j-\ell_k\log\ell_k\\
&\qquad -\log\left(1+\frac{1}{n}\right)^{Ln}\Bigg)\\
&=\ell_j+\delta\cdot\log\left(\left(1+\frac{\ell_k}{\ell_j}\right)^{\ell_j}\left(1+\frac{\ell_j}{\ell_k}\right)^{\ell_k}\left(1+\frac{1}{n}\right)^{-Ln}\right)\\
&=\ell_j+\Theta\left(\log\left(e^{\ell_k}e^{\ell_j}e^{-L}\right)\right)\\
&=\Theta(\ell_j+\ell_k-L).
\end{aligned}$$

If $\ell_j, \ell_k = O(L)$, the amortized cost is $O(L)$.

Load balancing operation

It remains to show that the expected amortized cost of a load exchange is negative. For this, we will need the assumption that $\varepsilon < 1/4$.

Let us first consider the case $i = j+1$. When moving items from node i to node j, the initial loads on those nodes are ℓ_i and ℓ_j, while both nodes will end up with a load of $(\ell_i+\ell_j)/2$. Thus, $(\ell_i-\ell_j)/2$ items have to be moved, which is the *actual* cost of the load exchange. The amortized cost therefore comes out to

$$\frac{\ell_i-\ell_j}{2}+\delta\left(2\frac{\ell_i+\ell_j}{2}\log\frac{\ell_i+\ell_j}{2}-\ell_i\log\ell_i-\ell_j\log\ell_j\right).$$

We have to show that this quantity is at most 0. For notational simplicity, let $\eta := \frac{\ell_j}{\ell_i} \leq \varepsilon$. Then we have the cost

$$\begin{aligned}
&\ell_i\frac{1-\eta}{2}+\delta\Big(\ell_i(1+\eta)\left(\log\ell_i+\log(1+\eta)-1\right)\\
&\qquad -\ell_i\log\ell_i-\eta\ell_i\log(\eta\ell_i)\Big)
\end{aligned}$$

$$
\begin{aligned}
&= \ell_i \left(\frac{1-\eta}{2} + \delta \Big((1+\eta)\log \ell_i + (1+\eta)\log(1+\eta) - (1+\eta) \right. \\
&\qquad\qquad \left. - \log \ell_i - \eta \log \eta - \eta \log \ell_i \Big) \right) \\
&= \ell_i \left(\frac{1-\eta}{2} + \delta \Big((1+\eta)\log(1+\eta) - \eta \log \eta - (1+\eta) \Big) \right) \\
&= \ell_i \left(\frac{1-\eta}{2} + \delta \left(\log(1+\eta) + \log \left(1 + \frac{1}{\eta}\right)^{\eta} - (1+\eta) \right) \right) \\
&\le \ell_i \left(\frac{1}{2} + \delta \left((\eta + 0.087) + \log \left(1 + \frac{1}{1/2}\right)^{1/2} - (1+\eta) \right) \right) \\
&\le \ell_i \,(0.5 - 0.12\delta)\,,
\end{aligned}
$$

using $\eta \le \varepsilon < 1/2$. Thus, for $\delta > 4.17$, we obtain that load exchanges are paid for by the drop in potential function.

The case $i \neq j+1$ involves the three nodes i, j and $j+1$. To simplify notation, we set $x := \ell_i$, $y := \ell_j$ and $z := \ell_{j+1}$. Recall that we have $y \le \varepsilon x$ and $z \le x$.

The actual number of items moved is $x/2 + y$. The three nodes' contribution to the potential function is $\delta(x \log x + y \log y + z \log z)$ before the update, and

$$
\delta \Big(\frac{x}{2} \log \frac{x}{2} + \frac{x}{2} \log \frac{x}{2} + (y+z)\log(y+z) \Big)
$$

after the update. So the change in potential function is

$$
\begin{aligned}
\Delta\Phi &= \delta \Big(2\frac{x}{2} \log \frac{x}{2} + (y+z)\log(y+z) - (x \log x + y \log y + z \log z) \Big) \\
&= \delta \Big(-x + (y+z)\log(y+z) - y \log y - z \log z \Big)\,.
\end{aligned}
$$

Note that the function $f(y,z) := (y+z)\log(y+z) - y \log y - z \log z$ is increasing in y and z for $y, z \ge 0$, since

$$
\frac{1}{\partial y} f = \log(y+z) - \log y \ge 0
$$

(and likewise $\frac{1}{\partial z} f \ge 0$ by symmetry).

Thus, the cost of the load balancing operation gets maximized for $y = \varepsilon x$, and $z = x$. The maximal amortized cost therefore is

$$
\begin{aligned}
&\frac{x}{2} + y + \delta \Big(-x + f(\varepsilon x, x) \Big) \\
&= \frac{x}{2} + y + x\delta \Big(-1 + (1+\varepsilon)\log((1+\varepsilon)x) - \varepsilon \log(\varepsilon x) - \log x \Big) \\
&= \frac{x}{2} + y + x\delta \Big(-1 + (1+\varepsilon)\log(1+\varepsilon) - \varepsilon \log \varepsilon \Big) \\
&= \frac{x}{2} + y + x\delta \left(-1 + \log(1+\varepsilon) + \log \left(1 + \frac{1}{\varepsilon}\right)^{\varepsilon} \right) \\
&\le x \left(\frac{3}{4} + \delta \left(-1 + \log \frac{5}{4} + \log \sqrt[4]{5} \right) \right) \\
&\le x\,(0.75 - 0.0975\delta)\,.
\end{aligned}
$$

This is less than 0 if $\delta > 7.7$. □

3.2 Discussion

The traffic caused by the update queries necessary for the protocol is sufficiently small that it can be buried within the maintenance traffic necessary to keep the P2P network alive. (Contacting a random node for load information only uses a tiny message, and does not result in any data transfers per se.) Of greater importance for practical use is the number of items transferred, which is optimal to within constants in an amortized sense.

The protocol can also be used if items are replicated to improve fault-tolerance, e.g. when an item is stored not only on the node primarily responsible for it, but also on the $O(\log n)$ following nodes. In that setting, the load ℓ_j refers only to the number of items for which a node j is *primarily* responsible. Since the item movement cost of our protocol as well as the optimum increase by a factor of $O(\log n)$, our scheme remains optimal within a constant factor.

3.2.1 Selecting Random Nodes

A crucial step in our protocol is the ability to be able to contact a random node in the P2P network. This is easy to achieve if the nodes are (almost) uniformly distributed in the address space: we pick an address uniformly at random, and then contact the node succeeding that address. The probability of selecting a node is equal to the fraction of address space spanned by it. It can be shown that our analysis still holds if all these probabilities are within a constant factor of $\frac{1}{n}$.

If the node distribution is skewed, then we have to employ a different scheme. One solution is that every node maintains an additional presence (i.e. virtual node) on the address ring. This virtual node is not used for storing items (i.e. does not take part in the item balancing protocol), its sole purpose is to be contacted for the random node selection. Using a balancing protocol such the one in Section 2 to distribute these virtual nodes evenly in the address space, we can again select a random node by choosing a random address and returning the virtual node following it.

Another solution for skewed node distributions is the use of the random skip list which we might also use for routing (see Section 3.2.3 below). Selecting a random element in such a data structure is a relatively straightforward task.

3.2.2 Weighted Items

In many circumstances, items stored in a P2P networks are not all equal in terms of the load they put on the hosting node. For example, items might have different sizes, so storing a larger items requires more disk space, or the popularity of items can differ, with some items being requested more often, leading to a higher I/O load on the hosting node.

We can model this by assigning a weight $w(x)$ to every item x. It turns out that our load balancing algorithm also works for the case where the load is defined as the sum of item weights, as opposed to the number of items stored at a node. There is one obvious restriction: load will be balanced only up to what the items themselves allow locally. As an example consider two nodes, one node storing a single item with weight 1, the other node a single item with weight 100. If these two nodes enter in a load exchange, then there is no exchange of items what will equalize the two loads.

Apart from this restriction, all the above analysis carries over to the weighted case, by simply treating an item with weight w as w items of weight 1.

Corollary 4 *Theorem 3 continues to hold for weighted items, with the following changes:*

(i) Load can be balanced only up to what the items' weights allow locally (see previous discussion).

(ii) The amortized total weight moved upon the insertion or deletion of an item with weight w is $O(w)$. □

3.2.3 Routing in Skewed Distributions

If the node distribution in the address space is very skewed, e.g. with very dense or adversarial clusterings of nodes, then Chord's routing is not guaranteed to take only $O(\log n)$ hops. Since for our protocol the node distribution mirrors the item distribution, this possibility cannot be excluded a priori. In such a case, an alternate routing infrastructure has to be created to retain the $O(\log n)$ hop routing guarantee. One possibility is to use a random skip list: the skip list is directed around the address ring, and nodes insert themselves at their active address with randomly chosen levels, where the probability of choosing higher levels decreases geometrically. For independent choices of levels, this retains the $O(\log n)$ whp routing guarantee of basic Chord, but works for arbitrary distributions of nodes in the address space.

3.2.4 Range Searches

Our protocol can provide load balance even for data that cannot be hashed. In particular, given an ordered data set, we may wish to map it to the $[0,1)$ interval in an order-preserving fashion. Our protocol then supports the implementation of a range search data structure. Given a query key, we can use Chord's standard lookup function (or the routing structure described in Section 3.2.3) to find the first item following that key *in the keys' defined order*. Furthermore, given items a and b, the data structure can follow node successor pointers to return all items x stored in the system that satisfy $a \leq x \leq b$. Since all nodes have a load of $\Theta(L)$, we obtain the first such protocol that achieves an $O(\log n + K/L)$ query time (where K is the size of the output).

4. CONCLUSION

We have given several provably efficient load balancing protocols for distributed data storage in P2P systems. Our algorithms are simple and easy to implement, so an obvious next research step should be a practical evaluation of these schemes. In addition, three concrete open problems follow from our work. First, it might be possible to further improve the consistent hashing scheme as discussed in Section 2.2. Second, it would be interesting to determine whether our item balancing protocol also works for the case where the cost of storing an item is node-dependent, e.g. because some nodes have greater storage capacity or bandwidth than others. And finally, our range search data structure does not easily generalize to more than one order. For example when storing music files, one might want to index them by both artist and year, allowing range queries according to both orderings. Since our protocol rearranges the items according to the ordering, doing this for two orderings at the same time seems difficult. A simple solution is to rearrange not the items themselves, but just store pointers to them on the nodes. This requires far less storage, and makes it possible to maintain two or more orderings at once. It is open how to solve this problem without such an added level of indirection.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments.

5. REFERENCES

[1] M. Adler, E. Halperin, R. M. Karp, and V. V. Vazirani. A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In *Proceedings STOC*, pages 575–584, June 2003.

[2] A. Anagnostopoulos, A. Kirsch, and E. Upfal. Stability and Efficiency of a Random Local Load Balancing Protocol. In *Proceedings FOCS*, pages 472–481, Oct. 2003.

[3] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings SODA*, pages 384–393, Jan. 2003.

[4] P. Ganesan and M. Bawa. Distributed Balanced Tables: Not Making a Hash of it all. Technical Report 2003-71, Stanford University, Database Group, Nov. 2003.

[5] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings IPTPS*, pages 242–250, Mar. 2002.

[6] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings VLDB*, pages 321–332, Sept. 2003.

[7] F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Hash Table. In *Proceedings IPTPS*, Feb. 2003.

[8] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Tools for Relieving Hot Spots on the World Wide Web. In *Proceedings STOC*, pages 654–663, May 1997.

[9] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings ASPLOS*, pages 190–201, Nov. 2000.

[10] D. M. Lewin. Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology, May 1998.

[11] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proceedings PODC*, pages 233–242, July 2002.

[12] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings PODC*, pages 183–192, July 2002.

[13] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings IPTPS*, pages 53–65, Mar. 2002.

[14] M. Naor and U. Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Proceedings SPAA*, pages 50–59, June 2003.

[15] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proceedings IPTPS*, Feb. 2003.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings ACM SIGCOMM*, pages 161–172, Aug. 2001.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[18] M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, Sept. 2003.

[19] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings SODA*, pages 331–340, Jan. 1993.

[20] R. G. Seidel and C. R. Aragon. Randomized Search Trees. *Algorithmica*, 16(4/5):464–497, Oct./Nov. 1996.

[21] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings ACM SIGCOMM*, pages 149–160, Aug. 2001.