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

David R. Karger[1] and Matthias Ruhl[2]

[1] MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
karger@csail.mit.edu
[2] 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.

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—for example, if we wish to support range-searches on "ordered" keys. 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.

1 Introduction

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 [1–4].

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 the 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 among nodes is not completely balanced. Some nodes end up with 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 is not adequate to balance the distribution of items among nodes. We give protocols to solve both of the load balancing challenges just described.

Address-Space Balancing. 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, many 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 [5], which requires $O(\log n)$ virtual copies to be operated for every node.

Virtual nodes have drawbacks. Besides increased storage requirements, they demand 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 particular item could take responsibility for that item's key and then refuse to serve the item.)

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 [1], we get 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 3—generally speaking, achieving (iii) required operating $O(\log n)$ virtual nodes, which pushed the degree to $O(\log^2 n)$ and failed to achieve (i).

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.

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

Preliminaries. We design our solutions in the context of the Chord DHT [4] but our ideas seem applicable to a broader range of DHT solutions. Chord 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 address assignment problem.

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

As discussed above, Chord maps items and nodes to a ring. We represent this space by the unit interval $[0, 1)$ with the addresses 0 and 1 are identified, so all addresses are a number between 0 and 1.

2 Address-Space Balancing

We will now 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 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 "virtual nodes"); it chooses exactly one of those virtual nodes to become *active* at any time—this is the only node that it actually operates. A node's set of virtual nodes depends only on the node itself (computed e.g. as hashes $h(i, 1), h(i, 2), \ldots, h(i, c \log n)$ of the node-id i), making malicious attacks on the network difficult.

We denote the address $(2b + 1)2^{-a}$ by $\langle a, b\rangle$, where a and b are integers satisfying $0 \le a$ and $0 \le b < 2^{a-1}$. 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 describe our protocol in terms of an ideal "locally optimal" state it wants to achieve.

Ideal state: Given any set of active virtual nodes, each (possibly inactive) virtual node "spans" a certain range of addresses between itself and the succeeding active virtual node. Each real node has activated the virtual node that spans the minimal possible (under the ordering just defined) address.

Note that the address space spanned by one virtual node depends on which other virtual nodes are active; that is why the above is a local optimality condition. Our protocol consists of the simple update rule that any node for which the local optimality condition is not satisfied, instead activates the virtual node that satisfies the condition. In other words, each node occasionally determines which of its $O(\log n)$ virtual nodes spans the smallest address (according to $\prec$), and activates that particular virtual node. Note that computing the "succeeding active node" for each of the virtual nodes can be done using standard Chord lookups.

Theorem 1. *The following statements are true for the above protocol, if every node has $c \log n$ virtual 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, when $\varepsilon \leq 1/2$ and $c \geq 1/\varepsilon^2$. (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.

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

Claim (ii) can be shown by arguing that every local improvement reduces the "distance" of the current state to the ideal state (in an appropriately chosen metric). We defer the details to the full version of this paper (cf [6, Lemma 4.5]), where we also discuss the rate at which local improvements have to be performed in order to guarantee load balance.

For the following, we will assume that virtual addresses are chosen independently at random. As with to the original consistent hashing scheme [7], this requirement can be relaxed to $\Omega(\log n)$-independence by applying results of [8]

To prove (iii), recall how we constructed the ideal state for claim (i) above by successively assigning nodes to increasing addresses. In this process, suppose we are considering one of the first $(1 - \varepsilon)n$ addresses. Consider the interval I of length ε/n preceding this address. At least εn of the real nodes have not yet been given a place on the ring. Among the possible $c\varepsilon n \log n$ possible virtual 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, the virtual 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 explains the additional claim.

For (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. Whenever a node claiming an address is deleted, its disappearance may reveal an address that some other node decides to claim, sacrificing its current spot, which may recursively trigger some other node to move. But each such migration means that the moving node has left behind no address as good as the one it is moving to claim. Note also that only a few nodes are close enough to any vacated address to claim it (distant ones will be shielded by some closer active node), and thus, as the address being vacated gets higher and higher in the order, it become less and less likely that any node that can take it will want it. We can show that after $O(\log\log n)$ such moves, no node assigned to a higher address is likely to have a virtual node close to the vacated address, so the movements stop. □

We note that the above scheme is highly efficient to implement in the Chord P2P protocol, since one has direct access to the address of a successor. 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 a $O(\log\log n/n)$ 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 $O(\log\log n)$ virtual nodes instead of just 1, we can reduce the fraction of items moved to $O(1/n)$ per node insertion, which is optimal within a constant factor. 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.

Related Work. Two protocols that achieve near-optimal address-space load-balancing without the use of virtual nodes have recently been given [9, 10]. 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.

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 virtual 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 addresses they might possibly occupy, 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 only insofar as it has to be able to contact "random" nodes in the system (in the full paper we show that this can be done even when the node distribution is skewed by the load balancing protocol). The protocol is the following (where ε is any constant, $0 < \varepsilon < 1$). 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 $\ell_i \leq \varepsilon \ell_j$ or $\ell_j \leq \varepsilon \ell_i$ 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 address intervals next to each other. Node j increases its address so that the $(\ell_i - \ell_j)/2$ items with lowest addresses 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 $j + 1$.

To state the performance of the protocol, 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 2. *If each node contacts $\Omega(\log n)$ other random nodes per half-life as well as whenever its own load doubles, then the above protocol has the following properties.*

(i) With high probability, the load of all nodes is between $\frac{\varepsilon}{8}L$ 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 $O(N/n)$ per node insertion or deletion. □

The proof of this theorem relies on the use of a potential function (some constant minus the entropy of the load distribution) that is large when the load is unbalanced. We show that item insertions and node departures cause only limited increases in the potential, while our balancing operation above causes a significant decrease in the potential if it is large.

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.

Extensions and Applications. 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.

Our protocol can be adapted for the case when items have weights, and the load of a node is the sum of the weights of the items stored at the node. The weight of an item can reflect its size, or its bandwidth consumption, in case of items with different popularity. The analysis is similar; we can show that the insertion or deletion of an item of weight w causes an amortized weight of $O(w)$ to be moved. There is however, one restriction: the protocol can only balance the load upto what the items allow locally. For 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.

The above 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 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 $a \leq x \leq b$. We give the first such protocol that achieves an $O(\log n + Kn/N)$ query time (where K is the size of the output).

Related Work. Randomized protocols for load balancing by moving items have received much attention in the research community. A P2P algorithm similar to ours was studied in [12]. However, their algorithm 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 [13], who also provide several further references. 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.

In recent independent work, Ganesan and Bawa [14] 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 without creating heavy network traffic for these nodes with extreme load.

Complex queries such as range searches are also an emerging research topic for P2P systems [15, 16]. An efficient range search data structure was recently given [17]. 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 that 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 $O(N + n \log n)$.

4 Conclusion

We have given several provably efficient load balancing protocols for distributed data storage in P2P systems. (More details and analysis can be found in a thesis [6].) Our algorithms are simple, and easy to implement, so an obvious next research step should be a practical evaluation of these schemes.

In addition, several concrete open problems follow from our work. First, it might be possible to further improve the consistent hashing scheme as discussed at the end of section 2. Second, 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. The drawback is that it requires another level of indirection, which might be undesirable for fault-tolerance reasons. Lastly, permitting nodes to choose arbitrary addresses in our item balancing protocol makes it easier for malicious nodes to disrupt the operation of the P2P network. It would be interesting to find counter-measures for this problem.

Acknowledgments

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

References

1. Kaashoek, F., Karger, D.R.: Koorde: A Simple Degree-optimal Hash Table. In: Proceedings IPTPS. (2003)

2. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In: Proceedings PODC. (2002) 183–192
3. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A Scalable Content-Addressable Network. In: Proceedings ACM SIGCOMM. (2001) 161–172
4. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proceedings ACM SIGCOMM. (2001) 149–160
5. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent Hashing and Random Trees: Tools for Relieving Hot Spots on the World Wide Web. In: Proceedings STOC. (1997) 654–663
6. Ruhl, M.: Efficient Algorithms for New Computational Models. PhD thesis, Massachusetts Institute of Technology (2003)
7. Lewin, D.M.: Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology (1998)
8. Schmidt, J.P., Siegel, A., Srinivasan, A.: Chernoff-Hoeffding bounds for applications with limited independence. In: Proceedings SODA. (1993) 331–340
9. Adler, M., Halperin, E., Karp, R.M., Vazirani, V.V.: A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In: Proceedings STOC. (2003) 575–584
10. Naor, M., Wieder, U.: Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In: Proceedings SPAA. (2003) 50–59
11. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the Evolution of Peer-to-Peer Systems. In: Proceedings PODC. (2002) 233–242
12. Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load Balancing in Structured P2P Systems. In: Proceedings IPTPS. (2003)
13. Anagnostopoulos, A., Kirsch, A., Upfal, E.: Stability and Efficiency of a Random Local Load Balancing Protocol. In: Proceedings FOCS. (2003) 472–481
14. Ganesan, P., Bawa, M.: Distributed Balanced Tables: Not Making a Hash of it all. Technical Report 2003-71, Stanford University, Database Group (2003)
15. Harren, M., Hellerstein, J.M., Huebsch, R., Loo, B.T., Shenker, S., Stoica, I.: Complex Queries in DHT-based Peer-to-Peer Networks. In: Proceedings IPTPS. (2002) 242–250
16. Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the Internet with PIER. In: Proceedings VLDB. (2003) 321–332
17. Aspnes, J., Shah, G.: Skip Graphs. In: Proceedings SODA. (2003) 384–393