New Algorithms for Load Balancing in Peer-to-Peer Systems

David R. Karger Matthias Ruhl

MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
{karger, ruhl}@theory.lcs.mit.edu

Abstract

Load balancing is a critical issue for the efficient operation of peer-to-peer networks. We give new protocols for several scenarios, whose provable performance guarantees are within a constant factor of optimal.

First, we give an improved version of consistent hashing, a scheme used for item to node assignments in the Chord system. In its original form, it required every network node to operate $O(\log n)$ virtual nodes to achieve a balanced load, causing a corresponding increase in space and bandwidth usage. Our protocol eliminates the necessity of virtual nodes while maintaining a balanced load. Improving on related protocols, our scheme allows for the deletion of nodes and admits a simpler analysis, since the assignments do not depend on the history of the network.

We then analyze a simple protocol for load sharing by movements of data from higher loaded to lower loaded nodes. This protocol can be extended to preserve the ordering of data items. As an application, we use the last protocol to give an efficient 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 peer to peer systems [1, 2, 3, 4, 5, 6, 7].

In general, distributed hash tables do not offer load balance quite as good as standard hash tables. A typical standard hash table evenly partitions the space of possible hash-function values; 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 hash-function range; 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 means 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 by each node for maintenance.

Below, we will discuss solutions to this problem that "reassign" (without any centralized computation) certain nodes to portions of the hash function range that are not adequately covered. In doing so, we run into an additional unusual constraint on our solution. Since a P2P system does not assume centralized control over the system, malicious behavior by certain nodes cannot be ruled out. As a form of protection against such attack, many P2P systems do not allow nodes to choose the portion of the key space for which they are responsible—if such a choice were possible, 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. Instead, responsibility is assigned in some "random" fashion that makes it less likely for a particular node to have control over specific data items. For example, Chord assigns each node to a portion of the key-space associated with a hash of the node's IP address.

Our goal of reassignment would seem to conflict with the goal of random assignment—it would seem that a malicious node could claim a need to be reassigned to some area it wants to control. To protect against this attack, we use limited reassignment: we show that key-space load balancing can be achieved by giving each node a set of $O(\log n)$ specific possible assignments (based, for example, on $O(\log n)$ distinct hashes of its IP address) and limiting its choice to which of those (few) assignments it accepts.

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 hypothetical key space is evenly distributed among the nodes (say, each given an even portion of the 0-1 interval), 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 elementary to prove that if nodes can only choose from a few assignments, then certain load balancing tasks are beyond them. Our solution to this problem therefor allows nodes to take on arbitrary assignments; with this freedom we show that we can load-balance an arbitrary distribution of items, without expending much cost in maintaining the load balance.

We design our solutions in the context of the Chord DHT [2] 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 used 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.

Notation. In this paper, we will use the following notation.

n = number of nodes in system

N = number of items stored in system (usually we have $N \gg n$)

a_i = number of items stored at node i

c_i = cost of storing an item at node i

$\ell_i = c_i \cdot a_i$ = weighted load on node i

$\overline{L} = N / \sum \frac{1}{c_i}$ = 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 Consistent Hashing

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 $O(\log n)$ other nodes to change their positions. Each node has a fixed set of $O(\log n)$ possible positions (called "slots") that it chooses from. This set only depends 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. The load-balanced state attained by our protocol is Markovian, i.e. it does not depend on the construction history.

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:

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

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

Ideal state: For each node i the following is true. For $j = 1,2,\ldots,c\log n$ let $\langle a_j,b_j\rangle$ be the minimal address (in terms of the just defined ordering $\prec$) in the interval between the slot $h(i,j)$ and the node following it in the address space. Then node i's address is $h(i,J)$ where $\langle a_J,b_J\rangle = \max\{\langle a_j,b_j\rangle \mid 1 \le j \le c\log n\}$. In case of a tie, slot $h(i,J)$ is chosen to be the slot *closest* to $\langle a_J,b_J\rangle$.

So in other words, each node picks the slot whose covered interval contains the minimal address. Our protocol consists of the simple update rule that any node for which the ideal state condition is not satisfied moves to its slot for which the condition is satisfied.

Theorem 1 *The following statements are true for the above protocol.*

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

(ii) In the ideal state of a network of n nodes, whp all neighboring pairs of nodes will be at most $5/n$ apart.

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

Proof Sketch: The ideal state can be constructed as follows. The node whose slot is closest below address 1 will be assigned to that slot. Then among the remaining nodes the one whose slot is closest below address $1/2$ will be assigned to that slot, and so on for increasingly longer addresses. For the first $n/2$ addresses we will whp assign a node to a slot at most $1/n$ below the address, since these intervals of size $1/n$ whp initially contain $\Omega(\log n)$ slots, of which a constant fraction belongs to nodes not yet assigned. Thus, neighboring nodes are at most $5/n$ apart. For (iii), by symmetry it suffices to consider a deletion. Whenever a node "assigned" to an address is deleted, the vacated spot will be filled by a random node among the ones assigned to higher addresses, which in turn vacates that higher address. By a simple analysis, this shows that whp only $O(\log n)$ movements occur. $\Box$

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.

Related Work. Two protocols that achieve near-optimal load-balancing without the use of virtual nodes have recently been given in [9, 10]. Our scheme improves upon them in two 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, and in fact load-balancing guarantees are only shown for the "insertions only" case.

3 Load Balancing by Moving Items

In this section, we consider a dynamic protocol that moves items from overloaded nodes to underloaded nodes. Unfortunately, this means that we cannot use Chord's lookup functionality to find items. This problem can be mitigated by using pointers to items, however the protocol is more suitable for contexts where "finding an item" is not a necessary operation, e.g. when items correspond to computational tasks.

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. The protocol is the following (where ε, $0 < \varepsilon < 1$, is some constant).

Load 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 node with higher load transfers items to the other node until their loads are equal.

To state the performance of the protocol, we need the concept of a *half-life*, which is the time it takes half the nodes, or half the items in the system to change.

Theorem 2 *If the costs c_i differ by at most a constant factor, then the above protocol has the following properties.*

(i) The load of all nodes is limited to at most $(\frac{4}{\varepsilon} - 2)\overline{L}$ whp, if each node contacts $\Omega(\log n)$ other random nodes per half-life or whenever its own load doubles.

(ii) Assuming these update rates, 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 traffic caused by the update queries necessary for the protocol is sufficiently small such that it can be carried out within the maintenance traffic necessary to keep the P2P network alive. (Note that contacting a random node only uses a tiny message, and does not result in any data transfers by itself.) Of a greater importance for practical use is the number of items to be transferred, which is $O(N/n)$ whp for any particular transaction, and optimal to within constants in an amortized sense.

Balancing ordered data. We can adapt the previous protocol to store ordered data, with the same performance guarantees within constant factors. In this setting, the items have an underlying ordering, and every node stores the items falling into a continuous segment of that ordering. This recovers the ability to support lookups of data items by key in $O(\log n)$ time. Furthermore, this protocol then allows for the implementation of 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).

Related Work. Work on load balancing by moving items can be found in [11]. Their algorithm is very similar to ours, however it only works for the static case, and they give no provable performance guarantees, only experimental evaluations.

Complex queries such as range searches are also an emerging research topic for P2P systems [12]. An efficient range search data structure was recently given in [13]. However, they do not address load balancing as an issue, 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 $O(N + n \log n)$.

4 Conclusion

We have given several provably efficient load balancing protocols for distributed data storage in P2P systems. (More details and analysis are in the full version of this paper [14].) Our algorithms are simple, and easy to implement, so an obvious next research step should be a practical evaluation of these schemes.

Further research on complex queries in P2P systems is likely to lead to challenging new load balancing questions, since the two problems seem to be closely correlated. This is because complex query data structures are likely to impose some structure on how items are assigned to nodes, and this structure has to be maintained by the load balancing algorithm.

References

[1] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings ACM SIGCOMM*, pages 161–172, August 2001.

[2] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings ACM SIGCOMM*, pages 149–160, August 2001.

[3] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, November 2000.

[4] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[5] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, March 2002.

[6] Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-optimal Hash Table. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[7] Dalia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings PODC*, pages 183–192, July 2002.

[8] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina 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] Micah Adler, Eran Halperin, Richard M. Karp, and Vijay V. Vazirani. A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks. In *Proceedings STOC*, pages 575–584, June 2003.

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

[11] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[12] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon T. Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings 2nd International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 242–250, March 2002.

[13] James Aspnes and Gauri Shah. Skip Graphs. In *Proceedings SODA*, pages 384–393, January 2003.

[14] David R. Karger and Matthias Ruhl. New Algorithms for Load Balancing in Peer-to-Peer Systems. Technical Report LCS-TR-911, MIT, July 2003.