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

July 16, 2003

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 several simple protocols for load sharing by movements of data from higher loaded to lower loaded nodes. These protocols 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.

Keywords: Peer-to-peer networks, load balancing, consistent hashing, searching ordered data.

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 peer to peer systems [RFH+01, SMK+01, KBC+00, RD01, MM02, KK03, MNR02]. 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. 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; the hash function is computed in a distributed fashion over the course of the routing hops.

A large number of algorithms have been proposed (and implemented in systems) [RFH+01, SMK+01, KBC+00, RD01, MM02, KK03] to provide distributed hash table functionality. The majority of them allow 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 [MNR02, KK03] that support $O(\log n)$-hop lookups with only constant neighbor degree; this is theoretically optimal but may be undesirable in practice due to fault-tolerance considerations.

An important issue in any P2P system is load balance: how many items get mapped to any one machine as compared to the average. 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 tight concentration of (total) load near the average. As an example, the Chord DHT is based upon consistent hashing [KLL+97], 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.

Below, we will 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 [SMK$^+$01] 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 (see section 2). 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.

A second interpretation of our results can be given independent of P2P systems. Consistent hash functions [KLL$^+$97] 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 $O(n \log n + m)$ space to store m items in n buckets. Our new scheme reduces the necessary space allocation to the optimal $O(n + m)$ space.

Our Contributions. In this paper we give three 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 whp, without use of virtual nodes. 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 that it chooses from. This set only depends on the node itself (computed e.g. as hashes of the node-id), 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.

Second, 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 moves keys from overloaded nodes to underloaded nodes. The 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. 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. It can be extended to the case where different nodes have different capacities and we wish to load balance in an appropriately scaled fashion. By itself, this protocol cannot be used for distributed data storage, as it allows items to be moved freely between nodes, making it impossible to find them. However, we will discuss several situations for which the scheme is useful, including the distribution of computational tasks or of large files that are pointed to from elsewhere.

In section 4, we adapt the previous protocol to store ordered data, with the same performance guarantees. 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, the last load balancing scheme 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. 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. Chord [SMK$^+$01] achieves load balances within a constant factor from optimum by employing consistent hashing [KLL$^+$97] and using $O(\log n)$ virtual nodes per real network node, incurring a corresponding overhead in routing.

Two protocols that achieve near-optimal load-balancing without the use of virtual nodes have recently been given in [AHKV03, NW03]. 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.

Work on load balancing by moving items can be found in [RLS$^+$03]. Their algorithm is very similar to the one given in section 3, 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 [HHH$^+$02]. An efficient range search data structure was recently given in [AS03]. 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 $O(N + n \log n)$.

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

Consistent Hashing [KLL+97] forms the basis of Chord's [SMK+01] assignment of items to nodes. In this scheme, both items and nodes are mapped to an address space $[0, 1)$ using some hash function (SHA-1 in the case of Chord). A node stores all items whose hash value is between the hash value of that node and the next higher node hash value. Unfortunately, in this basic scheme, the fraction of the address space assigned to a node can be up to $O(\log n/n)$. To achieve a maximal load of $O(1/n)$, in Chord every real node operates $\Theta(\log n)$ virtual nodes in the address space, leading to an overhead in storage and bandwidth usage.

In this section, we give a modification of the consistent hashing scheme that guarantees a maximal load of $O(1/n)$ by just using a single position for every node, improving space and bandwidth usage by a logarithmic factor.

We impose the additional restriction that each node i can only choose its address among a random, but fixed list of potential addresses $h(i, 1), h(i, 2), \ldots, h(i, c\log n)$, that we call "slots" for node i. Here h is some fixed hash-function mapping to the address space and c is some fixed constant. (In Chord, i is usually the IP-address of a node, but it can be any unique identifier.) The rationale behind this is to counter Byzantine attacks, where malicious nodes try to place themselves at a self-chosen address in the network in order to disrupt service. In our protocol, that capability of nodes is very much limited.

2.1 Insertions only: Splitting the largest interval

Before coming to the general dynamic case, we will briefly state a very simple protocol that splits the address space evenly if nodes are only inserted, but not deleted. It uses the following decision procedure to choose the address of a new node.

Largest interval: A node i considers for each of its slots $h(i, j)$ the address of the node closest before the slot (let that address be a_j^-) and address of the node closest after the slot (let that address be a_j^+). The node then moves to the slot $h(i, J)$ that lies within the largest interval, i.e. $a_J^+ - a_J^- = \max\{a_j^+ - a_j^- \mid 1 \le j \le c\log n\}$.

Lemma 1
After inserting n nodes according to the above protocol, whp every pair of neighboring nodes will be at most $56/n$ apart.

Clearly, the above bound can be tightened considerably. Simulations suggest that after inserting n nodes according to the above protocol with $c = 1$, the largest interval between neighboring nodes will be less than $\frac{2.5}{n}$.

Proof (Lemma 1): We want to bound the number of nodes we have to insert such that every address interval of the form $[i/n, (i+1)/n]$ for $i \in \{0, 1, \ldots, n-1\}$ will whp contain at least one node at the end of the insertion process. That will bound the distance between neighboring nodes to $2/n$.

Let us call these intervals "buckets", and let m_i be the number of empty buckets before inserting the i-th node (by empty bucket we mean that no node has been assigned to that address interval yet). Obviously, $m_1 = n$ and $m_{i+1} \in \{m_i, m_i - 1\}$.

We will split the insertion process into two phases. Let the first phase be all insertions while $m_i \geq n/(c \log n)$. During this phase, the probability that a node picks a slot in an empty bucket is at least

$$1 - \left(1 - \frac{1}{c \log n}\right)^{c \log n} \geq 1 - \frac{1}{e} > \frac{1}{2},$$

so at least a constant. So when inserting $26n$ nodes, this will happen at least $13n$ times whp, let us call these the "good" nodes.

However, even though some slot might land in an empty bucket, this does not imply that the slot in the *largest* bucket also falls in an empty bucket. We have to show that this bad case does not occur too often. For this consider only the good nodes for which their eventually chosen slot splits its interval at a ratio of $\frac{2}{3} : \frac{1}{3}$ or better (i.e. more evenly). This will be a third of the nodes in expectation, and $4n$ nodes whp.

If the largest interval hit by such a good node has size $\geq 1/n$, but the node does not hit an empty bucket, then the size of that interval is less than $3/n$. Such an interval can be split at most 3 times with a ratio of $\frac{2}{3} : \frac{1}{3}$ before the resulting sub-intervals each have size less than $1/n$. This bad case can therefore happen at most $3n$ times, showing that at least $4n - 3n = n$ of the good nodes actually end up in an empty bucket. So this first phase ends after at most $26n$ insertions.

Now to the second phase, where $m_i < n/(c \log n)$. This implies that the probability of hitting an empty bucket (or hitting any interval of size $\geq 1/n$) is *at most* a constant less than $2/3$. We show than another $O(n)$ insertions will fill all empty buckets whp. For this consider some arbitrary, initially empty bucket.

During $2n$ insertions, we choose $2cn \log n$ random slots, of which $c \log n$ fall within our empty bucket whp. Consider the nodes corresponding to these $c \log n$ random choices. For each of these nodes, the probability that any of its $c \log n - 1$ other random slots fell into another empty bucket is at most $2/3$. Thus, among all $c \log n$ of these nodes, whp at least one of them will *not* have chosen another slot in an empty bucket. This node will therefore move to the bucket, showing that it does not remain empty. Since the bucket was chosen arbitrarily, whp all of them will be filled after $2n$ insertions.

So in summary, inserting $28n$ nodes leads to a maximal load of $2/n$, or equivalently, inserting n nodes leads to a maximal load of $56/n$. $\square$

2.2 Insertions and Deletions

We now turn to the general dynamic case with both insertions and deletions of nodes. For the following, 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). Consequently we are going to speak about "shorter" and "longer" addresses to mean earlier or later in the ordering $\prec$.

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:

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

We are now going to describe our protocol in terms of the "ideal" state it wants to achieve. Later we will describe how this ideal state can be reached again after the insertion or deletion of elements.

Ideal state: For each node i the following is true. For $j = 1, 2, \dots, c\log n$ let $\langle a_j, b_j\rangle$ be the shortest address (in terms of the just defined ordering) 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 $\langle a_J, b_J\rangle = \max\{\langle a_j, b_j\rangle \mid 1 \leq j \leq c\log n\}$. 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 shortest 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.

We will shortly prove that such a system has a unique ideal state. This shows that the protocol is Markovian, i.e. is 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 where items are inserted with random priorities, yet the result does not depend on the order of the insertions.

Lemma 2
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 $4/n$ 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: Part (i) is easy to see, we can construct the ideal state as follows. Whichever node i has a slot $h(i, j)$ closest to (but less than) $\langle 0, 0\rangle = 1$, will move to that slot (note that this assignment necessarily happens in any ideal state). Among the remaining nodes, whichever node has a slot closest to $\langle 0, 1\rangle = 1/2$ will move there, and so on for the other addresses $\langle a, b\rangle$ in order of increasing length. Again, note that all these assignments are forced, so that there is only one ideal state. In the following we will say that a node was "assigned" to an address $\langle a, b\rangle$, if that address is the shortest address served by that node.

Note that some addresses $\langle a, b\rangle$ might be skipped in the assignment process, because all slots between $\langle a, b\rangle$ and the next shorter address preceeding it belong to nodes already assigned to shorter addresses. However, the longest address $\langle a, b\rangle$ that any node is assigned to has $a = O(\log n)$ whp.[1]

For part (ii) we will show that all addresses of the form $\langle a, b\rangle$ with $a \leq \lceil \log n \rceil - 1$ will have a node assigned to them whp. This implies that any two pair of neighboring nodes will be at most $2 \cdot 2^{-\lceil \log n \rceil - 1} \leq 4/n$ apart. For the following let $k := \lceil \log n \rceil$-1.

As mentioned previously, the only reason why we might skip assigning a node to an address $\langle k, b\rangle$ is because there are no slots in the interval $I := [2b \cdot 2^{-k}, (2b+1)2^{-k}]$ that do not belong to nodes assigned to shorter addresses. However, at most $n/2$ nodes are assigned to addresses shorter than $\langle k, b\rangle$, and the probability that none of the remaining $n/2$ nodes has a slot in I is at most

$$(1 - 2^k)^{cn/2 \log n} \leq (1 - 2/n)^{cn/2 \log n} \leq e^{-c \log n} = n^{-c}.$$

So with high probability there will be an unassigned slot in each of these intervals I, and therefore all addresses of the form $\langle a, b\rangle$ with $a \leq k$ will have nodes assigned to them, proving part (ii).

For part (iii), it suffices to analyze the deletion of a node, since the scheme is Markovian and therefore insertions and deletions are symmetric. Consider what happens when a node assigned to some address $\langle a_0, b_0\rangle$ gets deleted from the network. Then some element previously assigned to an address $\langle a_1, b_1\rangle \succ \langle a_0, b_0\rangle$ now gets assigned to $\langle a_0, b_0\rangle$, causing some element 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, until finally no element fills a vacated address. Since nodes only move to shorter addresses, the number of exchanges is clearly finite. We will now show that this number is $O(\log n)$ whp.

Let n_i be the number of elements assigned to addresses longer than $\langle a_i, b_i\rangle$. Note that the element moving to address $\langle a_i, b_i\rangle$ is uniformly at random among these n_i elements, thus $n_{i+1} \leq n_i/2$ with probability at least 1/2. As these "decrease by 1/2" events are independent, w.h.p. $O(\log n)$ movements suffice to reduce n_i to zero, showing that $O(\log n)$ nodes have to change their addresses upon the deletion or insertion of a node. $\square$

Practical Considerations. 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 (see appendix A). 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.

3 Load Balancing by Moving Items

In the remainder of this paper, we are going to describe and analyze algorithms that load balance a P2P network by actually moving the items stored on them. The advantage of this is that we can give load balancing guarantees even if the items are not uniformly distributed in the address space. The disadvantage is that we have to modify Chord's assignment of items to nodes to achieve this. That is, we either have to allow items or nodes to move freely within the address space.

[1]Consider addresses with $a = \lceil 4 \log n \rceil$, and the length 2^{-a} intervals preceeding them. The probability that any pair among the $cn \log n$ slots considered in the protocol falls in the same such interval is at most $(cn \log n)^2 \cdot 2^{-a} \leq (c^2 \log^2 n)/n^2$. So whp, this does not happen, but that means that every slot is closest to an address $\langle a', b\rangle$ with $a' \leq a$, and therefore gets assigned to such an address, as claimed.

To see why this is necessary, suppose every node is restricted to a small number of k possible locations (i.e. slots) (for example, $k = c \log n$ as above) in the address space $[0, 1)$. More precisely, we then cannot guarantee to handle distributions where an address interval of length p has more than a pk fraction of the load. This is easily shown by a counting argument: consider a set of $1/p$ non-overlapping intervals of length p. Then not all of them can contain more than npk slots, in fact one of these intervals will have at most npk slots. Thus, if we limit each node's load to $1/n$, we can only handle a load of pk in that interval.

In this section, we will consider the case where items can be moved to arbitrary nodes. Obviously this makes it impossible to use Chord's lookup mechanism to locate an item. This can be fixed by maintaining for every item a pointer at the primary node to the actual location of the item. This additional level of indirection is, while only changing the cost of operations by a factor of two, cumbersome in practice, and makes the system more error-prone.

The load balancing scheme is useful, however, for P2P applications where there is no need to "find" an item, for example if the items correspond to computational tasks, and we simply want to spread the computational load equally among the nodes. This problem will be solved by the protocol given in this section, even to the extent of making the "cost" of storing an item node-dependent, which can be used to model different speeds or storage capacities of nodes.

Another instance where destroying Chord's item-to-node mapping is not harmful is when one maintains enough structure in the load-balanced data to find items without resorting to Chord's built-in item location mechanism. We will show an example of this in the following section when dealing with ordered data.

Our load balancing protocol has a the following simple description (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 insert an item in this system, it is assigned to an arbitrary node for storage. When a node is inserted, it initially stores no items, and when a node is deleted, all its items are moved to arbitrary other nodes.

We will first analyze the load exchange protocol for the unweighted case, where $c_i = 1$ for all nodes i. In this case, the average load per node is $\overline{L} = N/n$.

For a simpler analysis, we will also assume for the moment that nodes j are chosen *uniformly* at random in the protocol. Later, we will discuss the difficulties with doing this in practice, and show how the protocol can be modified to work in an actual system, and how our analysis has to be changed in that case.

3.1 Analysis: The unweighted case

We will analyze the protocol both in terms of the load balance that it guarantees as well as the amount of data that has to move in order to achieve that load balance.

In traditional data structures problems, the typical model is for a number of modifications to the data structure as a result of invocations of the data structure operations—for example, a search tree may be rebalanced as a result of an item deletion. In peer to peer systems, with their unreliable participants, it is not generally possible to take such a reactive approach; instead, peer to peer systems must operate proactively even if no visible changes have occurred. As a concrete example, a general assumption in peer to peer systems is that nodes depart gracelessly, failing to inform other nodes of their departure. It is therefore necessary, as discussed by Liben-Nowell et al [LNBK02], for nodes in the systems to repeatedly attempt to contact their neighbors, to see if they remain alive, and to replace them if they have disappeared. That paper introduced the idea of evaluating the rate at which such contacts must be carried out, as a function of the rate at which nodes join and leave the system. That paper defined the *half-life* of a system to be time in which the half the nodes arrive or depart. We extend that definition to also measure the time in which half the items in a system are inserted or deleted.

We take a similar approach in analyzing our item-load balancing protocol. Rather than assuming that nodes will announce insertions and deletions, we assume a model in which nodes contact other nodes, at some rate, and ask what that rate must be as a function of the rate of insertion and deletion of nodes and items in order to maintain the desired load balance characteristics.

Lemma 3
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 $O(\overline{L})$ per node insertion or deletion.

Proof: For part (i), consider a node with large load. It suffices to show that in the time it takes a node's load to increase from $\frac{2}{\varepsilon}\overline{L}$ to $(\frac{4}{\varepsilon}-2)\overline{L}$, it will whp have contacted another node of load at most $2\overline{L}$ to split its load with, because that will bring the larger node's load again below $\frac{2}{\varepsilon}\overline{L}$. By Markov's inequality, at least half the nodes have load at most $2\overline{L}$, therefore it suffices to contact $\Theta(\log n)$ nodes to have a successful load exchange whp.

There are two possible reasons for a node's load to "increase" from $\frac{2}{\varepsilon}\overline{L}$ to $(\frac{4}{\varepsilon}-2)\overline{L}$. First, $(\frac{2}{\varepsilon}-2)\overline{L}=\Omega(\overline{L})$ items might have been added to the node, thus the node should query $\Omega(\log n)$ random nodes per item before its own load increases by a constant factor.

The second reason for a (relative) increase in load is if $\overline{L}$ drops by a constant factor. For this to happen, the number of items overall has to decrease by a constant factor or the number of nodes has to increase by a constant factor. So it suffices to contact $\Omega(\log n)$ nodes per half-life of the system.

For part (ii), we use a potential function Φ to keep track of the amortized costs. We define it as follows:

$$\Phi := \frac{\alpha}{\overline{L}}\sum_{i=1}^{n}\ell_i^2,$$

where $\alpha \geq \frac{1}{1-\varepsilon}$ is a constant. We bound the amortized costs of the insertions of items and nodes (deletions are symmetric), and of load exchanges by computing the quantities of "actual cost"$+\Phi_{\text{after}}-\Phi_{\text{before}}$.

Item insertion: When inserting an item, $\overline{L}$ will only become bigger. So the amortized cost upon insertion of an item into a node i is at most

$$\begin{aligned} O(1)+\Delta\Phi &\leq \frac{\alpha}{\overline{L}}\left((\ell_i+1)^2-\ell_i^2\right)+O(1)=\frac{\alpha}{\overline{L}}(2\ell_i+1)+O(1) \\ &\leq 2\alpha\left(\frac{4}{\varepsilon}-2\right)+O(1)=O\left(\frac{\alpha}{\varepsilon}+1\right)=O(1). \end{aligned}$$

Node insertion: Let $S := \sum_i \ell_i^2$. This quantity will not change during the insertion. We have $S < n(\frac{4}{\varepsilon}\overline{L})^2 = \frac{16}{\varepsilon^2}N^2/n$. So the amortized cost is:

$$\Delta\Phi = \frac{\alpha}{N/(n+1)}S - \frac{\alpha}{N/n}S = \frac{\alpha}{N}S \leq \frac{16\alpha}{\varepsilon^2}\frac{N}{n} = \frac{16\alpha}{\varepsilon^2}\overline{L} = O\left(\frac{\alpha\overline{L}}{\varepsilon^2}\right) = O(\overline{L}).$$

Load exchange: We have to show that when we actually perform a load balancing update, the change in potential function "pays" for the movement of the items. Among the two nodes involved in the exchange, let x be the load of the node with higher load, and y be the load of the other node, such that $y \leq \varepsilon x$.

The actual cost of the load balancing operation, i.e. the number of items moved, is $(x-y)/2$. The contribution of the two nodes to the potential function before the exchange was $(\alpha/\overline{L})(x^2+y^2)$ and becomes $(\alpha/\overline{L})(2((x+y)/2)^2)$ after the exchange. Thus, the amortized cost of the operation is:

$$\frac{1}{2}(x-y) + \frac{\alpha}{\overline{L}}\left(2\left(\frac{x+y}{2}\right)^2 - x^2 - y^2\right) = \frac{1}{2}(x-y) - \frac{\alpha}{2\overline{L}}(x-y)^2.$$

The amortized cost less than 0 if

$$\frac{1}{2} - \frac{\alpha}{2\overline{L}}(x-y) < 0 \Longleftrightarrow \frac{\overline{L}}{\alpha} < x - y$$

Since $(1-\varepsilon)x \leq x - y$ holds, a sufficient condition for the above is $\overline{L}/\alpha < (1-\varepsilon)x \Longleftrightarrow \frac{\overline{L}}{(1-\varepsilon)\alpha} < x$. As we consider a long sequence of such exchanges, it suffices if this holds in expectation. Since all elements participate in these updates, we can assume that in a random update $E[x] \geq \overline{L}$ (that would be true if x were picked at random, but if we condition on x being large enough to warrant an update, it makes $E[x]$ only larger). So the above holds if

$$E[x] \geq \overline{L} > \frac{\overline{L}}{(1-\varepsilon)\alpha} \Longleftrightarrow \alpha > \frac{1}{1-\varepsilon},$$

which is true if α is chosen as an appropriately large constant. □

Practical Considerations. 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 size message, and does not result in any data transfers per se.) Of a greater importance for practical use is the number of items to be transferred, which is $O(\overline{L})$ whp for any particular transaction, and optimal to within constants in an amortized sense.

3.2 Arbitrary probability distribution on nodes

So far, we have assumed that the nodes involved in an update are chosen uniformly at random. However, in an actual network, choosing nodes truly at random is a very hard, if not impossible task. So how do our protocol and analysis change if do not choose nodes uniformly at random? And how do we actually implement such a "random choice" in a P2P system, for example in Chord?

In Chord, we can use the following simple scheme: pick an address uniformly at random from the address space, and choose the node that serves that address. The probability of returning a node is then equal to the fraction of address space served by the node. Does our protocol still work in this case?

We used the fact that nodes are selected at random in two places in our analysis. First, in arguing the convergence of the protocol, we used the fact that in selecting a random node j, with probability at least $1/2$, the load of that node would be at most $2L$. Suppose in our new probability distribution, we can guarantee to pick with probability $1/2$ a node with load $\leq \delta\overline{L}$. Then a straightforward change of the analysis shows that the update rate given in Lemma 3 guarantees a maximal load of $\delta \cdot (\frac{2}{\varepsilon} - 1)\overline{L}$.

The second point in the analysis where randomness played a role is when we used $E[a_j] \geq \overline{L}$ in the proof of Lemma 3. Since all nodes use the load exchange protocol at similar rates, in every exchange one of the involved nodes will be a uniformly random node. Thus, the larger of the two still satisfies $E[a_j] \geq \overline{L}$, and the cost analysis does not have to change.

Thus, in order for the protocol to still "work", we must show that with probability at least $1/2$, a "randomly" picked node j satisfies $a_j \leq \delta\overline{L}$ for some constant δ. We then have the following Lemma.

Lemma 4
Given the update rates in Lemma 3, and an update rate that is high enough to guarantee $\Pr[a_i \leq \delta\overline{L}] \geq 1/2$ *at all times, the maximal load is limited to* $\delta(2/\varepsilon - 1)\overline{L}$ *at all times whp.* $\square$

Almost uniform probability distributions. An easy way to guarantee the performance of our protocol is to use some mechanism that makes each node serve at most a $O(1/n)$ fraction of the address space, e.g. the protocol described in section 2.2. Suppose each node serves at most a β/n fraction of the address space. This implies that for a randomly picked node j, we have $E[a_j] \leq \beta\overline{L}$, and thus by Markov's equality $\Pr[a_j \leq 2\beta\overline{L}] \geq 1/2$, and the condition "$a_j \leq \delta\overline{L}$" is satisfied with $\delta = 2\beta$.

3.3 Analysis: The weighted case

Suppose now that not all nodes are of equivalent ability, e.g. some nodes are faster, have more storage attached or a faster network connection, so it is easier for them to store data items. Or items actually represent programs, and nodes have different computation speeds. We model this by assigning every node i a value c_i, which represents the "cost" of storing a single item. So node i experiences a *weighted load* of $\ell_i = a_i c_i$. In this setting, we can show that our protocol balances the weighted loads if the costs c_i vary by at most a constant factor γ.

Note that balancing the load within a constant of optimum would be a trivial claim if that constant depended on γ, because weighted and unweighted loads are related within that factor. However, we are going to show a result where only the update rate and amortized item movement costs, but not the load guarantee, depend on γ. Recall that $\overline{L} = N/\sum \frac{1}{c_i}$.

Lemma 5
If the costs c_i differ by at most a constant factor, the load balancing protocol limits the weighted load of all nodes to within $(\frac{4}{\varepsilon} - 2)\overline{L}$ whp given the update rates in Lemma 3. Given these update rates, the amortized item movement costs are $O(1)$ per item insertion or deletion, and $O(N/n)$ per node insertion or deletion.

Proof: Let us consider the differences to the unweighted analysis. When selecting a node uniformly at random, with probability $\geq 1/(\gamma+1)$ it has a weighted load of at most $2\overline{L}$. To see this, note that the set of nodes S with weighted load of $2\overline{L}$ or more would store at most $N/2$ items if all nodes' loads were equal to $\overline{L}$. But that means that there can be at most γ times as many nodes in S than its complement (the nodes with load less than $2\overline{L}$). This gives the desired probability.

Since $1/(\gamma+1)$ is a constant probability, $O(\log n)$ trials suffice to find a node with load $2\overline{L}$ w.h.p. While the resulting load exchange is not guaranteed to bring a node's load below $\frac{2}{\varepsilon}\overline{L}$, it guarantees that it loses a constant fraction of its load. Therefore a constant number of such exchanges suffice to halve the load of the node. This shows the first claim of the Lemma.

Next, we have analyze the cost of the operations involved in load balancing. Again, we perform a amortized analysis using a potential function. This time, the potential function is

$$\Phi(\overline{a}) := \frac{\alpha'}{\overline{L}} \sum_{i=1}^{n} c_i a_i^2,$$

where α' is a constant to be chosen later (dependent on ε and γ).

First note that because we bounded the spread of costs c_i, we have $\overline{L} = \Theta(N/n)$, so the insertion and deletions of items and nodes can be argued in the same way as in the unweighted case. For exchanges, first note that the number of exchanged items is $m := (a_i c_i - a_j c_j)/(c_i + c_j)$, because

$$\begin{aligned}(a_i - m)c_i = (a_j + m)c_j &\iff a_i c_i - m c_i = a_j c_j + m c_j \\ &\iff a_i c_i - a_j c_j = m(c_i + c_j) \iff m = \frac{a_i c_i - a_j c_j}{c_i + c_j}.\end{aligned}$$

The drop in potential during an exchange is

$$\begin{aligned}&\frac{\alpha'}{\overline{L}}((a_i - m)^2 c_i + (a_j + m)^2 c_j - a_i^2 c_i - a_j^2 c_j) = \frac{\alpha'}{\overline{L}}(-2a_i m c_i + m^2 c_i + 2a_j m c_j + m^2 c_j) \\ = \; & \frac{\alpha'}{\overline{L}}(m^2(c_i + c_j) - 2m(a_i c_i - a_j c_j)) = \frac{\alpha'}{\overline{L}}(m^2(c_i + c_j) - 2m^2(c_i + c_j)) \\ = \; & -\frac{\alpha'}{\overline{L}} m^2 (c_i + c_j).\end{aligned}$$

For the amortized cost be at most 0, we have the following sufficient condition:

$$\begin{aligned}m - \frac{\alpha'}{\overline{L}} m^2 (c_i + c_j) \le 0 &\iff 1 \le \frac{\alpha'}{\overline{L}} m(c_i + c_j) = \frac{\alpha'}{\overline{L}}(a_i c_i - a_j c_j) \\ \Longleftarrow 1 \le \frac{\alpha'}{\overline{L}}(1 - \varepsilon) a_i c_i &\iff \frac{\overline{L}}{\alpha'(1 - \varepsilon)} \le a_i c_i.\end{aligned}$$

So by choosing α' large enough, this is satisfied in expectation. □

4 Balancing Ordered Data

So far, we have assumed that there is no structure on the data items, and that they can be arbitrarily assigned to different nodes in order to balance the load. In this section, we are going to assume that the items have an underlying order, and will give a load balancing algorithm that respects that ordering in a specified sense. For the following, we will assume that $c_i = 1$ for all nodes i.

As a motivating example, and a most useful algorithm in its own right, we will first state a protocol for range searching on ordered data, before discussing the load balancing protocol.

4.1 Range searching

The problem we solve in this section is the following. We want to store data items from some universe U in a distributed fashion in a P2P network. The universe U is equipped with a total ordering, denoted by "$<$". We want to support the following kind of queries on the set S of stored items.

Successor: Given a query $q \in U$, report $\min\{x \in S \mid q \leq x\}$.

Range search: Given $a, b \in U$ with $a < b$, report the set $\{x \in S \mid a \leq x \leq b\}$.

We will now describe a dynamic data structure with the following performance guarantees.

Successor: $O(\log n)$ nodes contacted in a query whp.

Range Search: $O(\log n + K/\overline{L})$ nodes contacted in a query whp, where K is the size of the answer.

Insert/delete item: $O(\log n)$ nodes to contact whp, and $O(1)$ amortized cost, in terms of number of items moved.

Insert/delete node: $O(\overline{L})$ amortized cost, in terms of number of items moved.

Load balance: Every node stores $\Theta(\overline{L})$ items whp.

We will first give the basic data structure without worrying about load balancing issues: The ordering on U is partitioned into non-overlapping intervals (this partition can change dynamically during the execution of our protocol). Each node is assigned one of these intervals, and stores all items from S that fall within that interval. The nodes are ordered according to the order of the intervals they are assigned.

On this linear sequence of nodes we maintain a skip list. When using Chord for the underlying routing infrastructure, we can give the nodes addresses in the order of their intervals, and use the Chord routing data structure as our skip list. Alternatively, we can maintain a randomized skip list as in [AS03].

Using the skip list, it is possible to find the node that stores the interval containing a query q by contacting $O(\log n)$ nodes. Either this node or its successor node (which can be found in one more step) stores the successor of q in S.

The implementation of the range search is also straightforward. First, we search for a's successor, and then output the elements at that node and the nodes following it until we reach the node that stores the interval containing b. This requires us to contact $O(K/\overline{L})$ nodes, assuming that every node stores $\Omega(L)$ nodes (cf. Lemma 6).

To insert or delete an item, we simply add it to or remove it from the node that is responsible for the interval containing it. To insert a node, we insert it at an arbitrary position of the address space, initially serving an empty interval. To delete a node, all its items get moved to the node following it, which then serves the union of the two nodes' intervals.

4.2 Balancing ordered data

The drawback of the above protocol is that it has no explicit mechanism for ensuring an evenly distributed load. We adapt the load balancing algorithm from section 3 for this purpose. While load balancing we will maintain the property that every node stores a contiguous segment of the ordering, and that the nodes are ordered according to the segments they store. The modified protocol is as follows, where $\varepsilon \in [0, 1/4)$ is again a constant (note that $\varepsilon < 1/4$ for this protocol).

Ordered load 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 store intervals next to each other. Node i transfers items *from the beginning of its interval* to node j so that both nodes end up with load $(\ell_i + \ell_j)/2$.

Case 2: $i \ne j+1$**:** If $\ell_{j+1} > \ell_i$, then we set $i := j+1$ and go to case 1. Otherwise, j first transfers all its items to its successor, node $j+1$. Then node i transfers the second half of its items to node j, and node j changes its position (address) to be between nodes i and $i+1$.

It is easy to see that the above scheme maintains the fact that the nodes store contiguous segments of the ordering. Its performance is very similar to the load balancing scheme from section 3, as the following lemma states. It is interesting to note however, that if the items are chosen from some probability distribution on U, then the interval partition will adapt to that distribution, and eventually become stable, requiring no more movements of items.

Lemma 6
The above protocol has the same performance guarantees as stated in Lemma 3, except:

(i) The protocol also bounds the load of all nodes from below by $\frac{\varepsilon}{4}\overline{L}$ whp, if each node contacts $\Omega(\log n)$ other random nodes per half-life or whenever its own load halves.

(ii) The amortized costs of item and node insertions and deletions increase by a constant factor.

Proof: The proof of Lemma 3 can be carried over almost verbatim (noting that in Case 2, $\ell_{j+1} > \ell_i$ does not happen too frequently). For the lower bound on the load per node, it suffices to show that a node with load $\le \frac{\varepsilon}{2}\overline{L}$ has a constant probability of contacting a node with load $\ge \overline{L}/2$. Since all nodes have load at most $\frac{4}{\varepsilon}\overline{L}$, the fraction of nodes with load $\ge \overline{L}/2$ must be at least $\frac{\varepsilon}{8}$, giving the desired constant probability. An analysis similar to Lemma 3 shows that the stated update rates suffice to have nodes with little load contact nodes with large load in order to receive items.

The amortized cost of the data movement is the same as in the proof of Lemma 3, except for the case of $i \ne j+1$. So let the potential function again be $\Phi = \frac{\alpha}{\overline{L}}\sum_i \ell_i^2$. 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 $\frac{\alpha}{\overline{L}}(x^2+y^2+z^2)$ before the update, and $\frac{\alpha}{\overline{L}}\left(\left(\frac{x}{2}\right)^2 + \left(\frac{x}{2}\right)^2 + (y+z)^2\right)$ after the update. So the change in potential function is

$$\Delta\Phi = \frac{\alpha}{\overline{L}}\left(2\left(\frac{x}{2}\right)^2 + (y+z)^2 - (x^2+y^2+z^2)\right) = \frac{\alpha}{\overline{L}}\left(-\frac{x^2}{2} + 2yz\right),$$

and the amortized cost comes out to

$$\frac{x}{2} + y - \frac{\alpha}{2\overline{L}}x^2 + \frac{2\alpha}{\overline{L}}yz \le x \cdot \left(\frac{1}{2} + \varepsilon + \frac{\alpha x}{\overline{L}}\left(-\frac{1}{2} + 2\varepsilon\right)\right).$$

Since we have $\varepsilon < 1/4$, the expected amortized cost is less than 0 iff

$$\frac{1}{2} + \varepsilon + \frac{\alpha x}{\overline{L}}\left(-\frac{1}{2} + 2\varepsilon\right) \le 0 \Leftrightarrow \overline{L}\frac{1/2+\varepsilon}{\alpha(1/2 - 2\varepsilon)} \le x$$

In a long sequences of exchanges, in expectation $E[x] \geq \overline{L}$, and therefore the above holds if

$$\frac{1/2 + \varepsilon}{\alpha(1/2 - 2\varepsilon)} < 1,$$

which can be achieved by choosing α as a large enough constant (depending on ε). □

5 Conclusion

We have given several provably efficient load balancing protocols for distributed data storage in P2P systems. The algorithms are simple, and easy to implement, so an obvious next research step should be a practical evaluation of these schemes. The given range search data structure improves on the previous best known structure [AS03] in reducing the query time from $O(\log N)$ to $O(\log n)$.

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

[AHKV03] 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.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *Proceedings SODA*, pages 384–393, January 2003.

[HHH+02] 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.

[KBC+00] 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.

[KK03] 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.

[KLL+97] 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.

[LNBK02] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the Evolution of Peer-to-Peer Systems. In *Proceedings PODC*, pages 233–242, July 2002.

[MM02] 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.

[MNR02] Dalia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings PODC*, pages 183–192, July 2002.

[NW03] Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. In *Proceedings SPAA*, pages 50–59, June 2003.

[RD01] 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.

[RFH+01] 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.

[RLS+03] 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.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kasshoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings ACM SIGCOMM*, pages 149–160, August 2001.

A Convergence of Consistent Hashing

Note that the deletion or insertion of a node requires a *linear* sequence of $O(\log n)$ local improvements. How can we make sure that these updates are performed in time to keep the system load balanced?

This is easy do to for *insertions*. The new node can easily find its correct address, which might cause the node previously assigned to that address to move, dislocating a further node, and so on, causing a linear sequence of moves. But since each affected node is aware of the fact that it has to move, these moves are easy to perform.

Much harder are *deletions*, even when nodes exit the network gracefully. This is because there does not exist a list of candidate nodes to serve the address vacated by the exiting node. We can solve this issue by having each node keep a list of the $\Theta(\log n)$ closest slots of other nodes following its address. These slots are considered when a node wants to leave the network. They can maintained efficiently, requiring only one update per node per half-life of the system. The "half-life" of a system is defined as follows, and gives a measure on how quickly a P2P system changes.

Definition 7 (Half-life)
The half-life *of a P2P system is the time it takes for half of the nodes in the system to change (by insertions or deletions).* □

This approach of remembering the $\Theta(\log n)$ closest slots is not of much use in the case where network nodes simply fail without invoking any deletion protocol. Similar to Chord, we therefore propose to run a background update process. Each node occasionally checks whether a local improvement is possible for itself, and if so, moves to the new address. If this displaces some node, that node in turn looks for a new position. Unfortunately, maintaining the ideal state by this update process would require a prohibitively large amount of traffic.

Conveniently, in order to keep the system load balanced, it is not really necessary to have the system in the ideal state, but merely sufficiently close to the ideal state. Recall the proof for Lemma 2. All that we needed for load balancing was that nodes were correctly assigned to the $n/2$ shortest addresses. This is much easier to achieve than having all n nodes be at their correct addresses.

Let a *round* be the time it takes until each node has performed a check whether it can do a local improvement. Then we can show the following sufficient condition on the number of rounds to achieve an almost ideal state.

Lemma 8
The following number of rounds are sufficient to assign a node to each address $\langle a, b\rangle$ with $b \leq \lceil \log n \rceil - 1$, within distance $\leq 2/n$ of that address (leading to a load balance of $4/n$).

(a) One round, if each node maintains the $\Theta(\log n)$ closest slots (by other nodes) after its active slot.

(b) $O(\log n)$ rounds without maintaining any additional information about other nodes.

Proof: Whenever a node checks for local improvements, it finds the nodes directly preceeding its slots, so maintaining the extra information of $\Theta(\log n)$ closest slots for each node is easy. This information can also be maintained if nodes fail, as long as one round is performed per half-life. This is because in one half-life, for each node whp only a $1/2+\varepsilon$ fraction of the closest slots belongs to failed nodes. So by choosing the constant hidden in the Θ-expression large enough, we can assure that we always have a sufficient number of live slots for the following. Note that maintaining this extra information does not change the $O(\log n)$ space requirement per node.

Let us call addresses $\langle a, b\rangle$ with $b \leq \lceil \log n \rceil - 1$ *short addresses*. We note that in the ideal assignment, whp one of the first $\Theta(\log n)$ slots following a short address will actually be occupied by a node.

For part (a), we can make the local improvement algorithm more efficient. Whenever a node moves to a different slot, belonging to a shorter address, it contacts the $\Theta(\log n)$ closest nodes, to see whether they want to claim the now free address. If this free address is a short address, then one of these nodes will actually be the node that is assigned to this address in the ideal assignment.

We now prove by induction, in increasing order of addresses, than all short addresses will have gotten assigned the correct node at the end of a round.

This is trivial for the shortest address 0, the node which should be assigned to it is the node with the address closest to it, and this node would notice that during its local improvement check, and move there.

Assume that we have proved that the assignment to the shortest i addresses is correct after the round. Consider the node x which should be assigned to this $(i+1)$-st address. To which address is it assigned after this round? Consider the last time that x performed a local improvement check. If it was assigned to an address among the first i addresses, then by induction hypothesis, it must later have been displaced from that slot, causing another local improvement check for x, contradicting that this was the last local improvement check. If it was assigned to an address longer than the $(i + 1)$-st address, then at that time a node was assigned to the $(i + 1)$-st address, which really should be among the first i addresses. However, x would have been remembered by that node and contacted when it finally moved its slot, causing it to move to the $(i+1)$-st address. This shows the claim for the $(i+1)$-st address.

Thus, by induction all short addresses have the correct node assigned to it. The important fact we used about the short addresses is that one of the first $\Theta(\log n)$ slots after it will actually become assigned to it. This is not necessarily true for longer slots, which is why we do not arrive at the ideal state, but merely at an approximation of it. The same will be true for claim (b).

For (b), we will show by induction on $k = 1, 2, \ldots, \Theta(\log n)$ that after k rounds all addresses for which the correct slot is among the first k slots after the address will have the correct node assigned to it. This is clearly sufficient to show the claim.

For $k = 1$, consider the set Q of nodes which will occupy the first slot after their addresses in the ideal state. When performing their local improvement update, they will either move to the correct address (since they displace whoever is currently assigned to that address), or to a shorter address. We claim that in the latter case it is later displaced from that address. Suppose not, then the node which should rightfully be at that address got assigned to a shorter address, and also not displaced there, and so on. We can thus construct an infinite chain of elements assigned to a too-short address, which is clearly impossible. Thus the element must be assigned to the correct slot.

It is not hard to see that this set of nodes Q will never move again in the following rounds. Thus we can remove it and its slots from consideration for the following rounds, which can therefore be argued in exactly the same way, showing the result for $k = 2, 3, \ldots, \Theta(\log n)$. $\square$