Peer-to-Peer Systems for Prefix Search

Baruch Awerbuch*
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
baruch@cs.jhu.edu

Christian Scheideler†
Department of Computer Science
Johns Hopkins University
3400 N. Charles Street
Baltimore, MD 21218, USA
scheideler@cs.jhu.edu

November 24, 2004

Abstract

This paper presents a general methodology for building a peer-to-peer data structure for prefix search for arbitrary user-defined names. The outcome of this is a peer-to-peer data structure that achieves (almost) the same performance as the best previously known peer-to-peer data structures for exact lookup. Our data structure achieves this result with the help of a plug&play paradigm that combines a data structure for prefix search with a data structure for shared memory management in such a way that their original performance is preserved. That is, we achieve a logarithmic search time for prefix search while ensuring that the data is evenly distributed among the peers, and maintaining this distribution is (almost) as cheap as in peer-to-peer systems for lookup. In order to prove the correctness of our plug&play paradigm and bounds on the work and expansion of our construction, we introduce a general, so-called spheres framework for the development of peer-to-peer data structures. The plug&play approach and the framework are very general and should therefore be of independent interest.

1 Introduction

Consider data items d with names (or identifiers) $\mathsf{Name}(d)$ out of some universe Names. The prefix search problem is to implement a tuple $(D, \mathsf{Insert}, \mathsf{Delete}, \mathsf{Search})$ consisting of a data set D and three operations Insert, Delete, and Search acting on D in the following way:

- $\mathsf{Insert}(d)$: this adds a data item d to the data structure, i.e. $D = D \cup \{d\}$. (We assume here that if there is already a data item d' in the data structure with $\mathsf{Name}(d') = \mathsf{Name}(d)$, then this gets replaced by d.)

- $\mathsf{Delete}(\mathsf{name})$: this removes the data item d with $\mathsf{Name}(d) = \mathsf{name}$ from the data structure, if it exists, i.e. $D = D \setminus \{d\}$.

- $\mathsf{Search}(\mathsf{name})$: this returns the data item d whose name is the closest prefix to the given name, i.e.

$$d = \mathsf{argmin}\{\mathsf{Name}(d') \mid d' \in D, \mathsf{Name}(d') \geq \mathsf{name}\}$$

  where "$\geq$" is with respect to lexicographical ordering. (If no such data item exists, "NULL" is returned.)

*Supported by NSF grant ANIR-0240551 and NSF grant CCR-0311795.
†Supported by NSF grant CCR-0311121 and NSF grant CCR-0311795.

These operations may be implemented so that they can be executed either sequentially or concurrently.

The three operations above are sufficient in order to implement a data structure for prefix search in systems with a static set of resources. However, in systems with a dynamic set of resources, such as peer-to-peer systems, we need two more operations: $\mathsf{Join}(p)$ and $\mathsf{Leave}(p)$. $\mathsf{Join}(p)$ allows a peer to join the data structure (with its resources), and $\mathsf{Leave}(p)$ allows a peer to leave the data structure (with its resources). To make sure that these operations do not create too much work, the data should be kept evenly distributed among the peers. Thus, a data structure for prefix search in peer-to-peer systems, also called a *peer-to-peer data structure* in the following, has to solve the following two tasks:

- How to organize the data so that the prefix search requirements are satisfied, and
- how to maintain an even distribution of the data items among the peers.

If only an exact lookup operation, instead of a prefix search operation, has to be provided, then the distributed hash table approach of Chord, CAN, Pastry, or Tapestry [26, 22, 24, 28] can be used to construct and maintain such a peer-to-peer data structure. In these approaches, an overlay network is maintained between the peers, and dynamic hashing is used to map data to peers in a load-balanced manner. For example, in Chord, every peer p obtains a hash value $h(p) \in [0, 1)$ and every data item d obtains a hash value $g(d) \in [0, 1)$, and every data item d is mapped to the peer p that currently has the lowest hash value with the property that $h(p) \geq g(d)$. Because hashing has the problem of destroying any relationship between the data, it appears to be difficult to extend this dynamic hashing approach to more advanced search problems such as prefix search. In fact, the task of constructing an efficient peer-to-peer data structure for prefix search was considered an open problem [7].

We solve this problem by proposing a plug&play approach that allows to decompose the problem of designing a peer-to-peer data structure for prefix search into two *independent* subproblems:

1. the problem of designing a data structure for prefix search, and
2. the problem of maintaining an even distribution of the data items among the peers.

Solutions to both problems already exist. For example, for the first problem, skip graphs [1] or hyperrings [3] may be used, and for the second problem, any peer-to-peer system providing a distributed hash table (e.g., [26, 22, 24, 28]) may be used. But nobody has previously managed to find out how to combine these results efficiently. A straight-forward approach would certainly be to provide a shared space platform with the help of a distributed hash table and then to map the nodes of the skip graph (pseudo-)randomly into the shared space as this is done for data items for exact lookup in distributed hash tables. That is, each node would be assigned to a fixed location in this shared space, and a node v establishes a connection to another node w in the skip graph by storing w's shared space location. However, the approach of combining two data structures in a *vertical* way, as would be the case with this shared space approach, has the problem that it significantly decreases the efficiency and expansion of the overall construction. For example, a prefix search operation in the skip graph requires $O(\log n)$ message transmissions, and each of these message transmissions requires $O(\log n)$ message transmissions within a distributed hash table such as Chord, resulting in an overall $O(\log^2 n)$ time bound for prefix search. In fact, in the original conference version of this paper, we suggested such an approach. In this paper, however, we propose a plug&play approach that allows to combine the two structures in a *horizontal* way, thereby preserving their original efficiency and expansion. In fact, as long as the number of data items in the system is at most polynomial in the number of peers, our approach achieves (almost) the same time and work bounds that were previously only known for peer-to-peer systems for exact lookup.

In order to achieve a horizontal combination of data structures, we propose a new computational framework for peer-to-peer systems, called the *spheres framework*. In its basic ideas, our approach is quite similar to the actors model designed by Carl Hewitt in the 1970s in the area of artificial intelligence [10]. Both the framework and our plug&play approach are very general and therefore should be applicable to many other problems besides prefix search.

Before we go into details concerning our approach, we conclude this section with a survey of previous work on search structures for peer-to-peer systems.

1.1 Previous work

In the following, a *concurrent* data structure denotes a data structure that allows the concurrent execution of operations. Concurrent data structures have been heavily investigated in the area of shared memory machines (e.g. [27, 6]). However, the problem with these data structures is that they only parallelize the *way* in which the data structure is accessed instead of parallelizing its *structure*. That is, the parallel search tree in [27] is still the tree, and the parallel skip list in [6] is still a skip list. This is acceptable in an environment with a static set of resources, such as parallel machines, but certainly not in an environment with a dynamic set of resources such as peer-to-peer systems. For peer-to-peer systems much more robust versions of concurrent data structures are needed. Ideally, a concurrent data structure for peer-to-peer systems should be *symmetric*, i.e. each node in it is equally important, and it should have a high expansion to make it very unlikely for it to become disconnected.

Several symmetric data structures for prefix search have recently been proposed. Randomized versions were found by Li and Plaxton [16], Aspnes and Shah [1], and Harvey et al [8], and deterministic versions were found by Harvey and Munro [9] and the authors [3]. The resulting data structures were named *skip graphs* [1], *skip nets* [8], or *hyperrings* [3], for example. They can all be seen as extensions of the randomized skip list data structure proposed by Pugh [21] to a concurrent environment. In [16, 1, 8] the focus is on proving upper bounds on the degree, path length, and expansion of these constructions. In addition to these, [3] also provides bounds on the congestion of routing concurrent search requests in the data structure.

None of the concurrent data structure proposals above consider the problem of how to embed the data structure into a dynamic peer-to-peer environment in a load-balanced way. The contribution of our paper is a paradigm accomplishing such an embedding, thus making the results of [16, 1, 8, 9, 3] applicable to peer-to-peer systems. Next, we describe the approach of [16, 1, 8, 9, 3].

Concurrent data structures for prefix search

All of the solutions in [16, 1, 8, 9, 3] are based on a doubly-linked list or cycle of data sorted in increasing order of their names. Each data item y has a pointer to its successor $z = \mathrm{succ}(y)$ and predecessor $x = \mathrm{pred}(y)$. This alone is not an ideal implementation, since in a list of n data items it can take $\Theta(n)$ steps to go from one data item to another. Moreover, the concurrent execution of search operations by n processes can cause a high congestion as a data item may be traversed by $\Theta(n)$ processes. There are several ways of adding shortcut pointers to alleviate this problem.

Perfect chordal graphs Consider a doubly-linked cycle in which each data item d keeps pointers to all data d' whose ranking in this cycle is exactly 2^i larger than its own, namely, $\mathsf{point}_i(d) = d'$ with

$$\mathrm{rank}(d') = \mathrm{rank}(d) + 2^i \pmod n$$

where $\mathrm{rank}(d) = r - 1$ if d has the rth smallest name. The pointer graph resulting from this belongs to the class of chordal graphs and is close to a hypercube. Furthermore, it can be shown that in a data structure as above with n data items stored along the cycle in the order of their names, n concurrent search operations with constant contention at the endpoints can be routed so that an only logarithmic amount of traffic passes through a single data item. However, maintaining perfect hypercubic pointers is very expensive since a single insert or delete operation requires to update $\Theta(n)$ pointers.

Skip list based approaches Consider the sequential skip list data structure initially suggested in [21] and extended to peer-to-peer environments in [16, 1, 8, 9, 3].

The basis of these data structures is a doubly-linked, sorted list ℓ of all data items. Imagine that we assign to each data item d an infinite bit string x_d. Then we can decompose ℓ into two sorted sublists: ℓ_0, which contains all sorted data d whose x_d starts with a 0, and ℓ_1, which contains all sorted data d whose x_d starts with a 1.

Now, we continue this process recursively on ℓ_0 and ℓ_1, generating even smaller sub-lists in the next higher level, namely ℓ_{00}, ℓ_{01}, ℓ_{10}, and ℓ_{11}, etc. Under the assumption that sub-lists are more or less of equal size, the recursion continues for a logarithmic number of levels. At level i of the recursion, a sublist $\ell_{b_1,b_2\dots b_i}$ is a doubly-linked list of all data d whose x_d has the prefix $b_1, b_2 \dots b_i$.

The following approaches of selecting the x_d's are possible.

- *Alternate bit padding:* Even-positioned nodes set their first bit to 0, and odd positioned nodes set their first bit to 1. Then ℓ_0 will contain even-positioned nodes and ℓ_1 will contain odd-positioned nodes. Continuing with this recursively, we obtain

$$\mathsf{point}_i(d) = d' \text{ where } \mathrm{rank}(d') = \mathrm{rank}(d) + 2^i$$

  i.e., we get a situation similar to the perfect chordal graphs above, with a high cost for insertions and deletions.

- *Random bit padding:* If the bit strings are random, then on expectation, $\mathrm{rank}(\mathsf{point}_i(d)) - \mathrm{rank}(d) = \Theta(2^i)$, and we get an approximation to a hypercube. The advantage of random choices is that its concurrent implementation of an Insert and Delete operation is much easier. For example, deleting data item y with predecessor x and successor z in a sublist $\ell_{b_1,b_2\dots b_i}$ involves simply changing x's (resp. z's) pointer point_i from y to z (resp. x). This is done for each one of the logarithmic number of levels at overall logarithmic cost. This is essentially the algorithm in [16, 1, 8].

- *Bounded bit padding:* Another way of selecting the bits is to require that there can be at most 2 consecutive nodes in any list (or cycle) at level i with the same $(i + 1)$th bit value. This property can be maintained with a low update cost and guarantees for every data item d and i that

$$\mathsf{point}_i(d) = d' \text{ with } \mathrm{rank}(d') = \mathrm{rank}(d) + O(2^i)$$

  This approach was taken in [9, 3].

Consistent Hashing: assignment of data to peers

Next we consider previous work on mapping data to a dynamic set of peers. Virtually all solutions presented so far (e.g., [26, 22, 24, 28, 17]) use consistent hashing. Consistent hashing was introduced by Karger et al. [13] as a means to manage the caching of web pages in a distributed environment. It works as follows:

Let V be the set of all possible web caches (or nodes) and D be the set of all possible data items. Consider the two hash functions $h : V \to [0, 1)$ and $g : D \to [0, 1)$ that map each node resp. data item to a real number in $[0, 1)$. Then the consistency condition that has to be maintained at all times is that each data item $x \in D$ currently in the system must be stored at the node $v \in V$ with minimum $h(v)$ so that $h(v) \geq g(x)$. If h and g are random or pseudo-random, it turns out that keeping the placement consistent is very cheap: (Pseudo-)random functions h and g ensure that at any time, the expected number of data items stored in a node is the same for all nodes. Hence, if the current number of nodes in the system is n, then the expected amount of data replacements when including or excluding a node is roughly a $1/n$ fraction of the total amount of data in the system.

The problem with using (pseudo-)random hash functions h and g is that this makes the data unsearchable in our sense, since hashing scrambles the name space, making prefix search impossible (i.e. only precise lookup operations can be supported). On the other hand, replacing hashing by a structure-preserving function g, such as the identity function, to enable prefix search may create a high load imbalance. This paper, however, will get the best of both worlds: functionality (i.e., prefix search capability) of [16, 1, 8] and the load balancing property of [26, 22, 24, 28, 17], thus solving the open problem in [7].

2 The Spheres Framework

In this section, we provide a formal framework for peer-to-peer computation that is necessary for our plug&play approach. The framework goes in parts far beyond the scope of this paper, and therefore not all aspects of the framework are discussed in detail here but only those relevant for the plug&play approach. A manuscript presenting the framework in full depth is currently in preparation [19].

As mentioned previously, the basic ideas behind this framework are very close to the ideas behind the actors model developed by Hewitt in the 1970s in the area of artificial intelligence [10]. Though his approach has never made it into a mainstream approach in distributed computing, it appears to be extremely useful for peer-to-peer computation. In fact, a small number of researchers has continued working on his ideas which led, among other results, to the language E [18]. E has some remarkable features, especially for robust and secure distributed computing, not shared by any other language and could be easily used to implement our framework. However, since this paper is about data structures for prefix search, going into implementation details here would be far beyond the scope of this paper. We refer instead the interested reader to www.erights.org.

2.1 The spheres framework

The spheres framework consists of three layers:

- **Network layer**: this is the lowest layer. It handles the exchange of messages between peers.
- **Contact layer**: this handles the exchange of messages between spheres and allows spheres to locate and interconnect to each other.
- **Spheres layer**: this is the layer for sphere-based applications.

In the network layer, any given communication mechanism may be used, such as TCP/IP, Ethernet, or 802.11. Its management is entirely an internal matter of the contact layer. Hence, the contact layer allows to hide networking issues from the spheres so that sphere-based applications can be written in a clean way. Thus, it remains to specify the spheres layer, the contact layer, and the interface between them.

2.2 The spheres layer

All computation and storage in the spheres layer is organized into so-called *spheres*. A sphere is an atomic thread with its own, private resources that are only accessible by the sphere itself. "Atomic thread" means that a sphere must be completely stored within a single peer and that operations within a sphere are executed in a strictly sequential, non-preemptive way. A prerequisite for this approach to work is that all elementary operations must be strictly non-blocking so that a sphere will never freeze in the middle of a computation.

A sphere cannot access any of the resources outside of its private resources. The only way a sphere can interact with the outside world is by sending messages to other spheres. A sphere is bound to the peer that created it. This is important to guarantee some basic properties for the exchange of messages between spheres specified below, but it does not restrict the universality of our approach.

2.3 The contact layer

All communication in the contact layer is handled via so-called *contact points*. Contact points can be thought of as ports but are much more general than that. A contact point is an atomic object that is bound to the sphere that created it. Given a sphere s, let $C(s)$ be the set of contact points created by it, and given a contact point c, let $R(c)$ be the set of spheres that have registered for it. Let *Contact* be the type of a contact point and *FarRef* be the type of a far reference. Contact points are manipulated by the following operations:

- *Contact* **NewContact**(): this creates and returns a new contact point c to the sphere s calling the operation. Formally, $C(s) = C(s) \cup \{c\}$.
- *FarRef* **Ref**(*Contact* c): this returns a far reference to contact point c.
- void **Register**(*Contact* c): this allows a sphere s to register for contact point c. Formally, $R(c) = R(c) \cup \{s\}$.
- void **Unregister**(*Contact* c): this allows a sphere s to unregister for contact point c. Formally, $R(c) = R(c) \setminus \{s\}$.

We demand that **Register** and **Unregister** operations from the same sphere to the same contact point are executed in FIFO order, i.e. in the order they were called, so that eventually $R(c)$ arrives at a correct state.

A contact point c does not need a destructor. As long as $R(c) = \emptyset$, c will not be stored explicitly but only if some sphere registers for it. Variables of type *Contact* or *FarRef* can be passed from one sphere to another. Ownership of a variable of type *Contact* allows to register and unregister for that contact point, and ownership of a variable of type *FarRef* allows to send a message to the contact point. For those familiar with capability-based systems, adding unguessable authentication information to these types turns them into capabilities so that contact points are protected against unauthorized access.

Given a contact point c with far reference r, the "$\longleftarrow$" operator allows to send a message m to any sphere that has registered for c, i.e. an anycast operation is performed. Or more precisely, if sphere s calls $\boldsymbol{r} \longleftarrow \boldsymbol{m}$, then the contact layer of s first checks whether s has already received a *representative* $s' \in R(c)$ from c. If not, s sends a request to c which returns any $s' \in R(c)$. Afterwards, s forwards the message to s'. If s' unregisters or becomes unavailable, the contact layer of s will request a new sphere from c.

The $\longleftarrow$ operation is a *non-blocking, eventual* send operator that guarantees the following properties:

- FIFO ordering: Messages sent by sphere s to some sphere s' via contact point c arrive at s' in the same order in which the $\boldsymbol{r} \longleftarrow \boldsymbol{m}$ operations were executed in s.
- At-least-once delivery: Messages are delivered to s' in an at least once fashion. (Notice that exactly-once delivery cannot be guaranteed in a potentially unreliable network.)

2.4 Creating and deleting spheres

Let $S(p)$ be the set of spheres that are currently at peer p. Spheres are created and deleted via the following operations.

- void **Spawn**($C(\cdot)$): this creates a new sphere s in the peer owning the sphere calling **Spawn**, using constructor $C(\cdot)$ for initializing s. Formally, $S(p) = S(p) \cup \{s\}$. Notice that no return value is provided. This makes sure that **Spawn** is non-blocking and the sphere calling **Spawn** has no initial access to s (which is useful for a formal security analysis).

- void **Halt**($D(\cdot)$): this unregisters the sphere s calling **Halt** from all contact points and waits until all tasks have been processed. (Notice that this happens in the background, i.e. the sphere will go on with executing code without waiting for the **Halt** operation to complete.) Afterwards, the destructor $D(\cdot)$ is called in s (which can be used, for example, to save the final state information of s or move it to another peer to recreate s there). Formally, $S(p) = S(p) \setminus \{s\}$.

For security reasons, the **Spawn** operation should only be allowed to be called by a special root sphere r which is created when the peer belonging to r initiates the platform providing the contact layer. In this way, the peer is under the control of which and how many spheres are spawned in it.

Notice that our model only requires seven primitives. All of these primitives are *necessary* in a sense that they cannot be replaced by a combination of the other operations. On the other hand, the primitives are also *sufficient* in a sense that they potentially allow a sphere to reside at any peer in the system and to communicate with any other sphere in the system. Thus, our framework is universal. Furthermore, it satisfies the demands we stated earlier on a distributed programming paradigm. Finally, notice its striking similarity with peer-to-peer systems: spheres are connected by a pointer structure in a similar way as peers are connected by an overlay network. In fact, the spheres framework allows to implement every peer-to-peer system that has been proposed so far, and much more.

Of course, one may wonder at this point whether the spheres framework will ever be more than just some thought experiment, but interestingly it is already much more than that. There is already a full-grown language support for it, and it allowed to solve an open problem about searchable peer-to-peer systems.

3 The Plug&Play Paradigm

In order to keep the presentation of our plug&play paradigm simple, we assume in the following that all peers and spheres behave correctly and depart gracefully, i.e. they use the Halt operation instead of just leaving the contact layer without notice. Of course, in the real world this is not the case, but dealing with these issues would certainly create an enormous overhead if not research problems by themselves. (In fact, not even a provably robust peer-to-peer system for exact lookup is currently known, apart from first steps in [5].)

In the following we describe how to use the spheres framework to combine two sphere structures in a transparent way in order to obtain a peer-to-peer data structure for prefix search. The two sphere structures we need are:

- a *site structure* for maintaining a load-balanced mapping of data items to peers and

- a *data structure* for organizing the data items in such a way that prefix search can performed efficiently in it.

The two sphere structures have to adhere to the following specifications.

4 The site structure

Let c_S be some predefined contact point for the site structure that every peer has access to when joining the contact layer. A site sphere s has to provide the following operations:

- Site(k_r, c_s): this constructs a site s. k_r is the far reference of some root sphere r that s can use to send requests to r and c_s is the contact point from which s can receive requests from r. To set this up, s just has to execute $\mathsf{Register}(c_s)$. s also calls Join.

- s.Join(): this lets s join the site structure by sending a corresponding request to contact point c_S. Once Join completes, s executes $\mathsf{Register}(c_S)$ to allow other sites to join the site structure.

- s.Leave(): this lets s leave the site structure. It starts with s calling $\mathsf{Unregister}(c_S)$ and ends with s calling Halt().

- s.Insert(d) for some data item d: this initiates the insertion of d into the site structure. (If there is already a data item in the system with the same name as d, the old data item is overwritten.)

- s.Delete(name): this initiates the deletion of the data item with name name in the site structure, if it exists.

Notice that a Search operation is not necessary because we will use the search functionality of the data structure to find data items. The site structure may implement the operations above in *any* way guaranteeing (for graceful departures) an even load distribution and the correctness of the operations under the assumption that the FIFO invariant holds.

Besides providing certain methods, site spheres also have to provide feedback to their roots. This happens in the following two cases:

- An Insert(d) request that started at site s reaches the site s' in which d is to be placed. Then s' invokes the method Forward(k_r, d) in r', where r is the root of s and r' is the root of s'. After the initial placement of d, this is also done each time d is moved from a site s to a site s'.

- A data item d is deleted in site s with root r. Then s invokes the method Remove(d) in r.

Notice that these are not artificial requirements. Even if there is only a site structure, a peer may want to know its current data load. The method invocations above are necessary for providing this feedback.

4.1 The data structure

Let k_D be some predefined contact point for the data structure that every peer has access to when joining the contact layer. A data sphere a has to provide the following operations:

- Data(k_r, c_a, d): this constructs a data sphere a. k_r is the far reference of some root r that a can use to send requests to r and c_a is the contact point from which a can receive requests from r. d is the data item a is supposed to represent. To set this up, a has to execute $\mathsf{Register}(c_a)$. a also calls Join.

- Data($k_{a'}, S$): this sets up data sphere a in state S. Once state S has been correctly set up (i.e. all relevant Register operations have been called), the Ack method is invoked in the data sphere reachable via $k_{a'}$.

- a.Join(): this lets a join the data structure by sending a corresponding request to c_D. Once Join completes, a executes $\mathsf{Register}(c_D)$ to allow other data spheres to join the data structure.

- a.Leave(): this lets a leave the data structure. It starts with a calling $\mathsf{Unregister}(c_D)$ and ends with a calling Halt().

- a.Search(name): this initiates the prefix search for name name. a remembers the reference k_r of its current root in case it moves during the search operation.

- a.Forward$(k_r, k_{r'})$: this requests a to move from root r to root r' (whose public keys are k_r and $k_{r'}$). If a is currently at root r, it invokes the Gone$((\mathsf{Name}(a), p(c_a)))$ method in r. Otherwise, it just stores $(k_r, k_{r'})$ in a set M for now and will check again for the invocation of Gone$((\mathsf{Name}(a), p(c_a)))$ once it is executing Data$(k_{a'}, S)$ in root r.

- a.Gone(): this is invoked by the Gone method in its root. a simply calls Halt(Frozen()) in this method.

- a.Frozen(): this initiates the movement for a. a picks the earliest method invocation of Forward$(k_r, k_{r'})$ in M it got for its current root r, packs its entire state (apart from c_a) into the state field $S(a)$, then calls c'_a = NewContact() and Register(c'_a) and finally invokes the Move$(p(c'_a), \mathsf{Name}(a), c_a, S(a))$ method in r' via $k_{r'}$.

- a.Ack(): this method is invoked once the new copy of a is ready to go. It simply consists of a executing Unregister(c'_a) and Halt().

The data structure may implement Join, Leave, and Search in any way guaranteeing the correctness of the operations under the assumption that the FIFO invariant holds.

Also data spheres provide feedback to their roots. Apart from the Gone method above, we require that whenever a data sphere a that initiated a Search(name) request on behalf of some root r got the data item d returned whose name is the closest prefix to name, then it invokes SearchDone(name, d) in that root r.

Certainly, using an sphere for every data item may be much too fine-grained if the data items are small. So we assume here for the moment that the data items are sufficiently large so that it is worth the effort. We will explain in Section 4.4 how to adapt our data structure approach so that it also works well for small data items.

4.2 Combining the site and data structure

Recall that every peer has a root sphere r that is the root of all spheres spawned in this peer. We will use this root sphere to combine any site structure and data structure fulfilling the specifications above into a peer-to-peer data structure for prefix search. r has a set of operations for this specified in Figure 1.

4.3 Correctness

Next we prove the correctness of the plug&play paradigm, i.e. we prove that

- the root spheres process the user-level operations according to their specification and that
- the root spheres correctly synchronize the site structure with the data structure.

The site structure and data structure are said to synchronize correctly if for every data item d whose insertion was initiated by site s_0 and that moved through sites $s_1, \ldots, s_k$ for some $k \geq 1$ before it got deleted in site s_k (or stays there forever), the data structure will eventually move the data sphere a representing d through the roots $r_0, \ldots, r_k$ and then remove a from the system (or keep it forever), where r_i is the root of s_i for every $i \in \{0, \ldots, k\}$.

We will use the following definitions for our correctness proof.

Definition 4.1 *For the site structure we define:*

1. *The Site constructor is completed once all operations in it are completed.*
2. *The* Join *operation of a site s is completed once s is fully integrated into the site structure.*

```
Root():
    c_r = NewContact()
    c_s = NULL

Join():
    if c_s = NULL then
        Register(c_r)
        c_s = NewContact()
        Spawn(Site(Ref(c_r), c_s))    // Ref(c): ref. of c
        D = ∅    // set of data spheres in r

Leave():
    if c_s ≠ NULL then
        Unregister(c_r)
        Ref(c_s) ← Leave()
        c_s = NULL

Insert(d):
    if c_s ≠ NULL then
        c_a = NewContact()
        Spawn(Data(Ref(c_r), c_a, d))
        d' = (Name(d),Ref(c_a))
        D = D ∪ {d'}
        Ref(c_s) ← Insert(d')

Delete(name):
    if c_s ≠ NULL then
        Ref(c_s) ← Delete(name)

Search(name):
    if c_s ≠ NULL then
        if D ≠ ∅ then
            k ← Search(Ref(c_r), name) for any d = (n, k) ∈ D
        else
            Ref(c_D) ← Search(Ref(c_r), name)

Move(k, name, c_a, S):    // data sphere requests copy in r
    Spawn(Data(k, S))
    D = D ∪ {(name, c_a)}

Forward(k_r', d):    // site: move d = (n, k) from r' to r
    k ← Forward(k_r',Ref(c_r))
    D = D ∪ {d}

Gone(d):    // data sphere of d = (n, k): gone
    D = D \ {d}
    k ← Gone()

Remove(d):    // site: remove d = (n, k)
    k ← Leave(Ref(c_r))
    D = D \ {d}

SearchDone(name, d):    // data sphere: search done
    deliver search result to user
```

Figure 1: The operations of the root sphere.

3. *The* Leave *operation of a site s is completed once s is completely removed from the site structure.*
4. *An* Insert(d) *operation is completed once d has been inserted into the site structure so that d is the only data item with name* Name(d).
5. *A* Delete(name) *operation is completed once there is no data item d in the site structure with* Name$(d) =$ name.

Definition 4.2 *For the data structure we define:*

1. *The Data constructor is completed once all operations in it are completed.*
2. *The* Join *operation of a data sphere a is completed once a is fully integrated into the data structure.*
3. *The* Leave *operation of a data sphere a is completed once a is completely removed from the data structure.*
4. *A* Search(name) *operation is completed once the data sphere that initiated the search reports the result back to its root.*

Based on these definitions, we can prove a sequence of claims. An operation of a root sphere is defined as being completed once all of the operations called in that operation are completed.

Claim 4.3 *Once the* Join *operation of a root sphere r is completed, r has a site sphere that is fully integrated into the site structure.*

Proof. By inspection of the Join operation in Figure 1 and Definition 4.1(1) and (2). □

Claim 4.4 *Once the* Leave *operation of a root sphere r is completed, r's site sphere is fully excluded from the site structure.*

Proof. By inspection of the Leave operation in Figure 1 and Definition 4.1(3). □

Claim 4.5 *Once the* $\mathsf{Insert}(d)$ *operation of a root sphere r is completed, r has a data sphere a for d that is fully integrated into the data structure and a data item $d' = (\mathsf{Name}(d), p(c_a))$ has been inserted into the site structure so that d' is the only data item with name* $\mathsf{Name}(d)$.

Proof. By inspection of the $\mathsf{Insert}(d)$ operation in Figure 1 and Definitions 4.1(4) and 4.2(1) and (2). □

Claim 4.6 *Once the* $\mathsf{Delete}(\mathsf{name})$ *operation of a root sphere r is completed, there is no data item d in the site structure with* $\mathsf{Name}(d) = \mathsf{name}$.

Proof. By inspection of the $\mathsf{Delete}(\mathsf{name})$ operation in Figure 1 and Definition 4.1(5). □

The claims above imply that the root spheres maintain a site structure that fulfills the specifications of a peer-to-peer data structure for prefix search w.r.t. Join, Leave, Insert, and Delete. Given that the data structure and the site structure synchronize correctly, this would also imply that the Search operation in the root returns a correct result when completed. Thus, it remains to show:

Claim 4.7 *The plug&play approach correctly synchronizes the data structure with the site structure.*

Proof. Consider any data item d. Suppose that the insertion of d (or rather the short form d' of d that uniquely identifies d through its name) was initiated at site s_0 and d went through sites $s_1, \ldots, s_k$ for some $k \geq 1$ before it got deleted at site s_k (or stays there forever). In the following, let r_i be the root of s_i for all $i \in \{0, \ldots, k\}$. The only situation in which s_0 initiates the insertion of a data item d is when the root r_0 of s_0 is executing Insert(d). But in this case d must have been part of the data structure when the Insert(d) operation completed. Once the site structure places d in s_i for some $i \geq 1$, s_i is required to invoke the Forward method in r_i with parameters $k_{r_{i-1}}$ and d. This in turn causes the invocation of Forward with parameters $k_{r_{i-1}}$ and k_{r_i} in the data sphere a of d. Thus, a receives move requests from r_{i-1} to r_i for all i. The order in which a receives these requests fulfills an important property:

For any site s for which there are $i_1, \ldots, i_\ell \in \{1, \ldots, k\}$ with $s = s_{i_j}$ for all $j \in \{1, \ldots, \ell\}$, the Forward method in a for s_{i_j} is invoked prior to the Forward method for $s_{i_{j+1}}$ for all $j \in \{1, \ldots, \ell - 1\}$.

This holds because of our FIFO Invariant: s reports movements to its root sphere r in the right order so that, according to the FIFO Invariant, they will also arrive in the right order at r. But if they arrive in the right order at r, then according to the FIFO Invariant, they will also arrive in the right order at a.

Hence, a knows the temporal relationship between move requests originating from the same site. This allows a to uniquely recover the path $s_0, s_1, \ldots, s_k$ the data item d went and therefore follow this path: when currently at root r, take the earliest move request received for r, if it exists.

Whenever a decides moving from root r to r', it invokes the Gone method in r which removes a from the list D of data items in r and causes r to close its connection from a by unregistering for c_a. Afterwards, r invokes the Gone method in a which causes a to freeze (by calling Halt(Frozen())). Once a is frozen, a transports its state over to root r' by invoking the Move method in it. Notice that once a is frozen, its contact point has no more messages to process and to send out, its root r will not send any further messages to a, and all messages destined for the contact points c for which $\mathsf{Out}(c) = \{a\}$ will be buffered at c. Hence, when the new copy a' of a is spawned at r' and the incoming connections are reestablished, a' can start receiving messages originally meant for a without the FIFO Invariant being violated when considering a and a' as a single sphere. Once a' reestablished all incoming connections, it invokes the Ack method in a causing a (and its contact point) to be removed from the system without losing any messages.

Finally, once s_k decides to delete d, it invokes the $\mathsf{Remove}(d)$ operation in its root r_k. r_k in turn invokes the $\mathsf{Leave}(\mathsf{Ref}(c_{r_k}))$ operation in sphere a. This tells a that the end of the path of d is at s_k, and therefore a can leave once it reaches s_k and there are no more movements to process.

Hence, the data structure and the site structure synchronize correctly w.r.t. any data item d, which completes the proof. □

4.4 A data structure for small data items

At the end of this section, we describe how one can establish a more course-grained representation of the data structure if the data items are small, i.e. it would be too much overhead to create a sphere for each data item. Instead, the following approach can be used:

Organize the data structure into spheres we call *buckets*. Each bucket is identified by a name range so that the name ranges of the buckets are pairwise disjoint and the union of the name ranges represents the entire name space. Each bucket has to contain between k and $4k$ data items, for some suitably chosen k. If a bucket has less than k data items, it merges with a neighboring bucket (we explain below how to do this), and if a bucket has more than $4k$ data items, it splits into two buckets by splitting its name space so that each of the two resulting buckets has the same amount of names. A bucket with name range $[n_1, n_2]$ is stored as a data item with name n_1 in the site structure. Thus, merging a bucket with another bucket means removing a data item from the site structure, and splitting a bucket means inserting a new data item into the site structure. Whenever a bucket merges or splits, the data spheres can notify their root spheres about this so that the root spheres can make sure that the corresponding data item is deleted from or inserted into the site structure. This in turn allows the site structure to send placement updates to the root structure (such as forwarding a data item from r to r') so that the root structure can place the buckets by invoking the Forward operation in them as this is done for the original data structure.

It remains to explain how to merge buckets. Here we can use a technique described, for example, in [4]: Suppose that the buckets are organized in a doubly-linked cycle in the order of their ranges. In order to break the symmetry of merge operations, we from groups of 2 or 3 buckets in the following way. Each bucket chooses a color $c \in \{black, white\}$. The following invariant has to be maintained for these choices:

Coloring Invariant: There are at most two neighboring buckets with the same color at any time.

In order to maintain this invariant, we use the following rules when a new bucket b joins the cycle: If b's neighbors have the same color, or one of b's neighbors has a neighbor with the same color, then b chooses the other color. Otherwise, b may choose any color. If a bucket b leaves, then its successor, b', checks whether afterwards the invariant is violated. If so, it changes its color.

The colors allow the buckets to be organized in disjoint groups of size 2 or 3 by using the rule that any consecutive group G of buckets starting with bucket b must fulfill that $c(b) = black$, the predecessor of b is white, and there are no two black buckets in G that are separated by a white bucket.

If a bucket in a group G declares it wants to merge, then it is checked whether it can merge with its neighbor in G without exceeding a capacity of $3k$. If so, it is merged (which causes a data item to be deleted in the site structure). Otherwise, the ranges are changed in the two buckets so that each of them has the same load (of at least $3k/2$, which causes an old data item to be deleted and a new data item to be inserted into the site structure). Merge operations are coordinated by the first bucket in a group.

The bucketing strategy above has two important benefits. First of all, it reduces the overhead of managing the data structure and also the site structure because now the site structure only needs to get involved in case of a merge or split event, reducing the number of method invocations in the site structure by a factor of k. But larger data spheres also imply fewer data spheres, and therefore a faster search time, which is usually the most important performance measure for peer-to-peer data structures for searching.

5 Performance of the Plug&Play Paradigm

Next we analyze the performance of our plug&play paradigm. We start with work bounds and then prove bounds on the expansion of our approach.

5.1 Work

In order to bound the work of our operations, it suffices to bound the number of contact-layer operations necessary to execute any of the user-level operations of a peer-to-peer data structure for prefix search using our plug&play approach. This is indeed sufficient because all communication is handled by the contact layer. Since each contact layer operation can be implemented with a constant communication work (given that the messages and references have a constant size), our contact-layer bounds will provide us with upper bounds on the total communication work for our operations.

In the following, let n be the current number of site spheres and m be the current number of data spheres in the system. We assume for simplicity that $m \geq n$. Let $w_o^s(n,m)$ denote an upper bound on the work to execute operation o in the site structure and $w_o^d(m)$ denote an upper bound on the work to execute operation o in the data structure. The work here counts all methods invoked from an actor in the specified structure. That is, a Forward invocation of a site sphere counts towards the work in $w_{Join}^s(n,m)$, but not the subsequent methods triggered by it because Forward is invoked in the root of the site. So the work bounds for the site structure represent the amount of data movements and message transmissions necessary to execute the operation within that structure. We assume that data is only moved in the site structure when a site joins or a site leaves it, which is the case for all distributed hash tables. Then the following work bounds can be obtained for our plug&play approach.

Theorem 5.1 *Let $\Delta(m)$ be the maximum degree of an actor in the data structure. For our plug&play approach it holds:*

- Join *needs $O((w_{Join}^d(m) + w_{Leave}^d(m))w_{Join}^s(n,m))$ work,*
- Leave *needs $O((w_{Join}^d(m) + w_{Leave}^d(m))w_{Leave}^s(n,m))$ work,*
- Insert *needs $O(w_{Insert}^s(n,m) + w_{Join}^d(m))$ work,*
- Delete *needs $O(w_{Delete}^s(n,m) + w_{Leave}^d(m))$ work, and*
- Search *needs $O(w_{Search}^d(m))$ work.*

Proof. We need the following lemma.

Lemma 5.2 *Every operation that a site sphere invokes in a root sphere creates at most $O(w_{Join}^d(m) + w_{Leave}^d(m))$ work.*

Proof. A site sphere can only invoke two operations in its root r: Forward and Remove. Forward causes r to invoke the Forward method in the corresponding data sphere a, which eventually moves to the new root r'. When it decides to move, a chain of operations will be invoked: r.Gone, a.Gone, a.Halt(Frozen()), a.Frozen, a.NewContact, a.Register, r'.Move, r'.Spawn, a.Ack, and finally a.Unregister and a.Halt. The work in r'.Spawn is upper bounded by $w^d_{Join}(m)$. Thus, summing everything up, we obtain a work of $O(w^d_{Join}(m))$. The Remove operation invokes a.Leave whose work is upper bounded by $w^d_{Leave}(m)$. Hence, combining the two work bounds results in the lemma. □

A data sphere can only invoke the operations Gone and SearchDone in its root, but Gone is already accounted for in the Forward operation above and SearchDone only delivers the search result. So we can ignore the work for these operations.

Since Join (resp.) Leave can trigger at most $w^s_{Join}(n,m)$ (resp. $w^s_{Leave}(n,m)$) data movements, the bound in the theorem follows from Lemma 5.2. Insert requires the integration and movement of a single data sphere and the insertion of a data item into the site structure whose work is bounded by $O(w^s_{Insert}(n,m) + w^d_{Join}(m))$. Delete requires removing a data sphere from the data structure and removing a data item from the site structure whose work is bounded by $O(w^s_{Delete}(n,m) + w^d_{Leave}(m))$. Finally, the Search operation only needs $O(w^d_{Search}(m))$ work because it only affects the data structure. □

The work bounds above also translate into time bounds if we assume that it takes (at most) one unit of time to execute one unit of work. However, measuring the work is usually more general than measuring time since in asynchronous systems it is not guaranteed that operations finish within a certain amount of time.

Let us look now at an example to illustrate the bounds above. If we choose the Koorde network as the site structure, then it follows from results in [12] that for any $o \in \{\text{Join}, \text{Leave}\}$, $w^s_o(n,m) = O(\sigma \cdot m/n + \log n)$ with high probability, and for any $o \in \{\text{Insert}, \text{Delete}\}$, $w^s_o(n,m) = O(\log n)$ with high probability. The reason for the high work bounds for Join and Leave is that up to $O(\sigma \cdot m/n + \log n)$ data items may have to be moved, with high probability, due to a Join or Leave operation. The parameter σ denotes the imbalance of the data distribution and can be bounded by $O(\log n)$, with high probability, for Koorde [12]. Using better distribution schemes (e.g. [14, 23]), one can even achieve a constant σ without increasing the degree of the nodes.

Using a skip graph [1] as the data structure, it follows that for any $o \in \{\text{Join}, \text{Leave}, \text{Search}\}$ that $w^s_o(m) = O(\log m)$ with high probability. Thus, when recalling our assumption that $m \geq n$, we obtain the following result.

Corollary 5.3 *Using Koorde as the site structure and the skip graph as the data structure, it holds that, with high probability,*

- Join *needs* $O(\sigma \cdot (m/n) \log m)$ *work,*
- Leave *needs* $O(\sigma \cdot (m/n) \log m)$ *work,*
- Insert *needs* $O(\log n + \log m)$ *work,*
- Delete *needs* $O(\log n + \log m)$ *work, and*
- Search *needs* $O(\log m)$ *work.*

As long as m is polynomial in n, this asymptotically (almost) matches the best bounds that were previously known only for peer-to-peer data structures for exact lookup.

5.2 Expansion

Finally, we bound the expansion of our construction.

Definition 5.4 *Given a graph $G = (V, E)$ and a subset $U \subseteq V$, the* expansion *of U is defined as $\alpha(U) = |\Gamma(U)|/|U|$ where $\Gamma(U)$ is the set of nodes in $V \setminus U$ that have an edge from U. The* expansion *of G is defined as*

$$\alpha(G) = \min_{U, |U| \leq |V|/2} \alpha(U) .$$

We will investigate the expansion of horizontal peer-to-peer data structures, which is our approach, as well as the expansion of vertical peer-to-peer data structures, which was the approach of the conference version of this paper. "Horizontal" in our context means that the pointer structure of the data structure is used independently of the site structure, and "vertical" means that the data structure is completely embedded into the site structure, i.e. for every edge (v, w) that a request wants to traverse in the data structure, the request has to travel along a path in the site structure from the site of v to the site of w.

Recall that our plug&play approach uses two sphere structures: a data structure, denoted by D, and a site structure, denoted by S. Given any fixed sequence J of name insertions and removals, let $\alpha_D(J)$ be the median expansion of the pointer graph of D after executing J, i.e. the probability that the pointer graph of D has an expansion of at least resp. at most $\alpha_D(J)$ at the end is a least $1/2$. Then the expansion $\alpha_D(m)$ of D is defined as

$$\alpha_D(m) = \max_{J \in \mathcal{J}_m} \alpha_D(J)$$

where $\mathcal{J}_m$ is the set of all sequences J that end up with m data items.

We define the expansion of the site structure in a similar way. Given any fixed sequence K of site insertions and removals, let $\alpha_S(K)$ be the median expansion of the pointer graph of S after executing K, i.e. the probability that the pointer graph of S has an expansion of at least resp. at most $\alpha_S(K)$ at the end is a least $1/2$. Then the expansion $\alpha_S(n)$ of S is defined as

$$\alpha_S(n) = \max_{K \in \mathcal{K}_n} \alpha_S(K)$$

where $\mathcal{K}_n$ is the set of all sequences K that end up with n sites.

We need one more parameter for S called the *imbalance* that is relevant for horizontal and vertical peer-to-peer data structures. Given a fixed mapping of data items to sites, let $L(s)$ be the load (i.e. the number of data items) at site s. Then the *imbalance* $\sigma(S)$ of S is defined as

$$\sigma(S) = \max_s \frac{L(s)}{m/n}$$

where m is the number of data items and n is the number of sites.

Now, let H be the peer-to-peer data structure resulting from the horizontal composition of D and S and V be the peer-to-peer data structure resulting from the vertical composition of D and S. We introduce the following definitions of the expansion of H and V:

Definition 5.5 (Horizontal expansion) *Consider the horizontal composition H of D and S. For any subset D' of data in D and any subset S' of sites in S we say that S'* blocks *D', denoted as $S' \oslash D'$, if for every $d \in D'$ it holds for a majority of data $d' \in D$ that for every path p in D from d' to d there is at least one datum d'' in p that lies in S'. Then we define the expansion of H as*

$$\alpha(H) = \mathrm{E}\left[\min_{S' \oslash D'} \frac{|S'|/|S|}{|D'|/|D|}\right]$$

where the expected value is over the uniform distribution of locations of the data items in the shared space of the site structure.

Intuitively, the horizontal expansion gives the expected worst case ratio between the percentage of sites (resp. peers) that are down and the percentage of data spheres in D that would be disconnected by this from the rest of D.

Definition 5.6 (Vertical expansion) *Consider the vertical composition V of D and S. For any subset D' of data in D and any subset S' of sites in S we say that S'* blocks *D', denoted as $S' \oslash D'$, if for every $d \in D'$ it holds for a majority of sites $s \in S$ that for every path p in D from a datum d' in s to d there is at least one edge (v, w) in p for which there is no path from v to w in S when removing S'. Then we define the expansion of V as*

$$\alpha(V) = \mathrm{E}\left[\min_{S' \oslash D'} \frac{|S'|/|S|}{|D'|/|D|}\right]$$

where the expected value is over the uniform distribution of locations of the data items in the shared space of the site structure.

Thus, as for the horizontal expansion, the vertical expansion gives the expected worst case ratio between the percentage of sites (resp. peers) that are down and the percentage of data spheres in D that would be disconnected by this from the rest of D.

First we bound the horizontal expansion, and afterwards we bound the vertical expansion.

Theorem 5.7 (Horizontal Composition Theorem) *For any data structure D and site structure S, it holds for their horizontal composition H that*

$$\alpha(H) = \Omega\left(\frac{1}{\sigma(1 + \frac{1}{\alpha_D})}\right)$$

where σ is an upper bound on the imbalance that holds with high probability.

Proof. Consider any fixed horizontal composition H, i.e. the imbalance of H and the expansion of D are fixed and denoted by $\sigma(H)$ and $\alpha(D)$. For a subset D' (S') of data (sites) denote by $||D'||$ ($||S'||$) the relative fraction of such data (sites) i.e., the ratio $|D'|/|D|$ (respectively, $|S'|/|S|$). Let Λ_S be the set of sites removed from S and let Λ_D be the set of data in Λ_S. Then it follows from the definition of $\sigma(H)$ that

$$|\Lambda_D| = \sum_{s \in \Lambda_S} L(s) \leq \sigma(H) \cdot \frac{|D|}{|S|} \cdot |\Lambda_S|$$

and therefore

$$||\Lambda_D|| \leq \sigma(H) \cdot ||\Lambda_S||$$

Furthermore, according to the definition of $\alpha(D)$, $|\Lambda_D|$ many data spheres can disconnect at most $|\Lambda_D|/\alpha(D)$ many other spheres from the majority of the data spheres. Hence, the set of data Γ_D outside of Λ_D that are inaccessible from a majority of data spheres fulfills

$$||\Gamma_D|| \leq ||\Lambda_D||/\alpha(D)$$

Thus, it follows for $\alpha(H)$ that there is a set Λ_S so that

$$\alpha(H) = \frac{||\Lambda_S||}{||\Gamma_D|| + ||\Lambda_D||} \geq \frac{1}{\sigma(H)(1 + \frac{1}{\alpha(D)})}$$

Now, notice that by definition, $\Pr[\alpha(D) \geq \alpha_D] \geq 1/2$. Hence,

$$\Pr\left[\frac{1}{1+\frac{1}{\alpha(D)}} \geq \frac{1}{1+\frac{1}{\alpha_D}}\right] \geq \frac{1}{2}$$

This in turn implies that

$$\mathrm{E}\left[\frac{1}{1+\frac{1}{\alpha(D)}}\right] \geq \frac{1}{2} \cdot \frac{1}{1+\frac{1}{\alpha_D}}$$

Furthermore, we know that $\sigma(H) = \Theta(\sigma)$, w.h.p., and therefore $1/\sigma(H) = \Omega(1/\sigma)$, w.h.p. Since $\sigma(H) \geq 1$, it also follows that

$$\mathrm{E}\left[1/\sigma(H)\right] \leq \left(1 - \frac{1}{n^k}\right) \cdot \frac{c}{\sigma} + \frac{1}{n^k} \cdot 1$$

for some constants c and k. Hence, $\mathrm{E}[1/\sigma(H)] = \Theta(1/\sigma)$. Plugging the bounds for $\mathrm{E}[1/\sigma(H)]$ and $\mathrm{E}[1/(1+1/\alpha(D))]$ into the inequality above results in the bound for $\alpha(H)$ in the theorem. $\square$

It follows from [3], for example, that the hyperring has an expansion of $\Omega(1/\log m)$. Hence, when using the hyperring as the data structure, we obtain an expansion of $\Omega(\log n \cdot \log m)$ for H, which can be improved to $\Omega(\log m)$ when using techniques in [14, 23]. If m is polynomial in n, this would match the best bounds known for the expansion of distributed hash tables, which is $\Omega(1/\log n)$ for Chord, for example [2].

Theorem 5.8 (Vertical Composition Theorem) *For any data structure D and site structure S, it holds for their vertical composition V that*

$$\alpha(V) = \Omega\left(\frac{1}{\sigma \cdot (1+\frac{1}{\alpha_S})(1+\frac{1}{\alpha_D})}\right)$$

where σ is an upper bound on the imbalance that holds with high probability.

Proof. Consider some fixed vertical composition V, i.e. $\sigma(V)$, $\alpha(D)$ (the expansion of D), and $\alpha(S)$ (the expansion of S) are fixed. Let Λ_S be the set of sites removed and Γ_S be the set of sites in $S \setminus \Lambda_S$ inaccessible in S from a majority in S. (I.e. for every site $s \in \Gamma_S$ there are more than $|S|/2$ sites that are disconnected from s when removing Λ_S.) Furthermore, let Υ_S be the set of sites that can reach the majority in S. For a subset D' (S') of data (sites) denote by $||D'||$ ($||S'||$) the relative fraction of such data (sites) i.e., the ratio $|D'|/|D|$ (respectively, $|S'|/|S|$).

Since S has an expansion of $\alpha(S)$,

$$||\Gamma_S|| \leq \frac{||\Lambda_S||}{\alpha(S)} \,. \tag{1}$$

Now, let Λ_D be the set of data stored at Λ_S and Γ_D be the set of all data in $D \setminus \Lambda_D$ that are inaccessible in D from a majority in S. (I.e. for every datum $d \in \Gamma_D$ there are more than $|S|/2$ sites $s \in S$ so that for every path p in D from a datum d' in s to d there is at least one edge (v, w) in p for which there is no path from v to w in S when removing S'.) Also, let Υ_D be the set of those data which are accessible in V from a majority in S. Note that the sets Λ_D, Υ_D, and Γ_D are disjoint in D.

Next let us decompose $\Gamma_D = \Gamma_D^1 \cup \Gamma_D^2$ such that $\Gamma_D^1 \cap \Gamma_D^2 = \emptyset$ and Γ_D^1 contains all data in Γ_D stored at sites in Γ_S. By the definition of $\sigma(V)$ and inequality (1), the fraction of such files is at most

$$||\Gamma_D^1|| \leq ||\Gamma_S|| \cdot \sigma(V) \leq \frac{||\Lambda_S||}{\alpha(S)} \cdot \sigma(V)$$

Now, consider any datum $d \in \Gamma_D^2$, and suppose that d is stored at site s. By definition, $s \notin \Lambda_S$ (since then $d \in \Lambda_D$) and also $s \notin \Gamma_S$ since then we would have added d to Γ_D^1. So $s \in \Upsilon_S$. Suppose now that d has a neighbor d' with $d' \in \Upsilon_D$. Then, by definition, the site s' of d' must be in Υ_S. But if both s and s' are in Υ_S, then there is a path from s to s', which implies by the definition of Υ_D that also $d \in \Upsilon_D$, a contradiction.

Hence, every neighbor of a datum $d \in \Gamma_D^2$ that is not in Γ_D^2 must be in $\Lambda_D \cup \Gamma_D^1$. From above we know that

$$||\Lambda_D \cup \Gamma_D^1|| \leq ||\Lambda_S|| \cdot \sigma(V) + \frac{||\Lambda_S||}{\alpha(S)} \cdot \sigma(V) = \sigma(V) \cdot \left(1 + \frac{1}{\alpha(S)}\right) \cdot ||\Lambda_S||$$

From the definition of $\alpha(D)$ it therefore follows that

$$||\Gamma_D^2|| \leq \frac{\sigma(V)}{\alpha(D)}\left(1 + \frac{1}{\alpha(S)}\right)||\Lambda_S||$$

Summing it all up, $||\Gamma_D||$ is equal to

$$\begin{aligned} ||\Gamma_D^1|| + ||\Gamma_D^2|| &\leq \sigma(V) \cdot \frac{||\Lambda_S||}{\alpha(S)} + \frac{\sigma(V)}{\alpha(D)}\left(1 + \frac{1}{\alpha(S)}\right)||\Lambda_S|| \\ &= \sigma(V) \cdot ||\Lambda_S|| \cdot \left(\frac{1}{\alpha(D)} + \frac{1}{\alpha(S)} + \frac{1}{\alpha(D) \cdot \alpha(S)}\right) \end{aligned}$$

According to the definition of $\alpha(V)$, there is a set Λ_S so that

$$\alpha(V) = \frac{||\Lambda_S||}{||\Gamma_D|| + ||\Lambda_D||}$$

Using the fact that $||\Lambda_D|| \leq \sigma(V) \cdot ||\Lambda_S||$, it follows that

$$\frac{||\Lambda_S||}{||\Gamma_D|| + ||\Lambda_D||} \geq \frac{1}{\sigma(V)(1 + \frac{1}{\alpha(D)})(1 + \frac{1}{\alpha(S)})}$$

Now, notice that by definition, $\Pr[\alpha(D) \geq \alpha_D] \geq 1/2$ and $\Pr[\alpha(S) \geq \alpha_S] \geq 1/2$. Hence, it follows from the proof of the previous theorem that

$$\mathrm{E}\left[\frac{1}{1 + \frac{1}{\alpha(D)}}\right] = \Omega\left(\frac{1}{1 + \frac{1}{\alpha_D}}\right) \quad \text{and} \quad \mathrm{E}\left[\frac{1}{1 + \frac{1}{\alpha(S)}}\right] = \Omega\left(\frac{1}{1 + \frac{1}{\alpha_S}}\right)$$

Furthermore, it follows from the previous proof that $\mathrm{E}[1/\sigma(V)] = \Theta(1/\sigma)$. Plugging these bounds into the inequality above and using the fact that $\alpha(D)$ and $\alpha(S)$ are independent results in the bound for $\alpha(V)$ in the theorem. □

The two composition theorems demonstrate, not surprisingly, that horizontal designs should be preferred over vertical designs because they are more robust.

6 Conclusions

In this paper we showed how to develop a peer-to-peer data structure for prefix search that (almost) achieves the same performance that was previously only known for peer-to-peer data structures for exact lookup. We introduced a new framework for the development of peer-to-peer data structures and a plug&play approach for combining two independent data structures that are very general and therefore should be of independent interest. Even if the reader may not be completely satisfied with our spheres framework, we hope that our approach will lead to further, exciting work in developing a general, well-accepted framework for peer-to-peer systems in the future that will allow rigorous, formal correctness and performance proofs for peer-to-peer data structures and that is close enough to reality so that data structures that are efficient and robust in this framework are also efficient and robust in real life.

7 Acknowledgements

We would like to thank the reviewers for their many critical and helpful comments on an earlier draft of this paper. We also thank Andreas Terzis for many discussions on peer-to-peer systems and Mark Miller for introducing us to the E language and helping us with the formulation of a general-purpose framework.

References

[1] J. Aspnes and G. Shah. Skip graphs. In *Proc. of the 14th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pp. 384–393, 2003.

[2] B. Awerbuch and C. Scheideler. Chord++: Low-congestion routing in Chord. Unpublished manuscript, 2003. See http://www.cs.jhu.edu/~scheideler.

[3] B. Awerbuch and C. Scheideler. The Hyperring: A low-congestion deterministic data structure for distributed environments. In *Proc. of the 15th ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pp. 318–327, 2004.

[4] B. Awerbuch and C. Scheideler. Consistent and compact data management in distributed storage systems. In *Proc. of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 44–53, 2004.

[5] B. Awerbuch and C. Scheideler. Group Spreading: A protocol for provably secure distributed name service. In *Proc. of the 31st International Colloqium on Automata, Language, and Programming (ICALP)*, pp. 183–195, 2004.

[6] J. Gabarro, C. Martnez, and X. Messeguer. A design of a parallel dictionary using skip lists. *Theoretical Computer Science*, 158:1-33, 1996

[7] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[8] J.A. Harvey, M.B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS 2003*.

[9] N. J. Harvey and I. Munro. Deterministic SkipNet. *Information Processing Letters*, 90(4):205–208, 2004.

[10] C. Hewitt, P. Bishop, and R. Stieger. A universal modular Actor formalism for artificial intelligence. In *Proc. of the 1973 International Joint Conference on Artificial Intelligence*, pp. 235–246.

[11] W. Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, 58:13–30, 1963.

[12] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[13] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the 29th ACM Symposium on Theory of Computing (STOC)*, pp. 654–663, 1997.

[14] D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. of the 16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 36–43, 2004.

[15] K. Lakshminarayanan, A. Rao, I. Stoica, and S. Shenker. Flexible and Robust Large Scale Multicast Using I3. UCB Technical Report No. UCB/CSD-02-1187, May 2002.

[16] X. Li and C.G. Plaxton. On name resolution in peer-to-peer networks. In *POMC 2002*, pages 82–89.

[17] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 183–192, 2002.

[18] M.S. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. *Financial Cryptography 2000*. See http://www.erights.org/elib/capability/ode/.

[19] M.S. Miller and C. Scheideler. Spheres: A new/old model for distributed computing. Manuscript, Johns Hopkins University, October 2004.

[20] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. In *Proc. of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 50–59, 2003.

[21] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *WADS 1989*, pages 437–449.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM 2001*.

[23] C. Riley and C. Scheideler. Local load balancing in distributed hash tables. Technical report, Johns Hopkins University, February 2004. See http://www.cs.jhu.edu/∼scheideler.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware 2001*.

[25] C. Scheideler and W. Wei. A load-balanced peer-to-peer registration service and its applications to anycasting and multicasting. Manuscript. Johns Hopkins University, September 2004. Available from the authors on request.

[26] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM 2001*.

[27] U. Vishkin, W. J. Paul, and H. Wagener. Parallel dictionaries on 2-3 trees. In *10th International Colloqium on Automata, Languages, and Programming (ICALP)*, pp. 597–609, 1983.

[28] B.Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A Global-scale Overlay for Rapid Service Deployment. *IEEE Journal on Selected Areas in Communications, Special Issue on Service Overlay Networks*, 2003.