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

ABSTRACT

This paper presents a general methodology for building message-passing peer-to-peer systems capable of performing prefix search for arbitrary user-defined names. Our methodology allows to achieve even load distribution, high fault-tolerance, and low-congestion concurrent query execution. This is the first known peer-to-peer system for prefix search with such properties. The essence of this methodology is a plug and play paradigm for designing a peer-to-peer system as a modular composition of arbitrary concurrent data structures.

1. INTRODUCTION

1.1 Distributed searchable data structures

Consider data items o with names (or identifiers) $\mathsf{Name}(o)$ out of some universe Names. A standard "uni-processor" *searchable data structure* $\mathcal{U}$ is accessed through a single handle (entry point) h into $\mathcal{U}$ (e.g., a tree root or the front of a queue). It supports the following standard operations:

- $h.\mathsf{Insert}(o)$: adds data item o to $\mathcal{U}$.
- $h.\mathsf{Delete}(\mathsf{key})$: removes the data item with name key from $\mathcal{U}$.
- $h.\mathsf{Search}(\mathsf{key})$: returns the closest prefix o^* in $\mathcal{U}$ to key from above: $o^* = \mathsf{argmin}\{\mathsf{Name}(o) \mid o \in \mathcal{U}, \mathsf{Name}(o) \geq \mathsf{key}\}$ where "$\geq$" is with respect to lexicographical ordering.

Fault-tolerant data structures [2] and concurrent data structures useful for a network of processors may also have multiple handles h into the data structure.

In order to implement a searchable data structure in a distributed, dynamic environment, we also need operations for the reorganization of the data placement if new participants (or sites) enter the system or old participants leave the system. Hence, in addition to supporting Insert, Delete, and Search, a *distributed searchable data structure* $\mathcal{D}$ must also support

- $s.\mathsf{Join}(s')$ contacts site s' in $\mathcal{D}$ to integrate new site s into $\mathcal{D}$.
- $s.\mathsf{Leave}()$ removes site s from $\mathcal{D}$.

If only precise lookup is required, then the distributed hash table approach of Chord, CAN, Pastry, Tapestry, or Viceroy [13, 11, 12, 15, 8] can be used to construct and maintain a distributed data structure. In these approaches, a connected pointer graph (or overlay network) is maintained between dynamically changing sites, and hashing is used to map data to sites in a load-balanced manner. While no impossibility result has been formally shown, it appears to be difficult to accomplish prefix search. In fact, the task of constructing distributed searchable data structures was considered an open problem [4].

We solve this problem with a new plug-and-play paradigm: instead of designing yet another distributed data structure, we show how to combine existing (non-searchable or imbalanced) concurrent data structures (see Section 1.2) to construct a searchable (and balanced) distributed data structure.

We describe the general principle of this paradigm in Section 2 and give implementation details in Section 3. In Section 4, we introduce parameters measuring the fault-tolerance and the ability to route requests with low congestion in order to quantitatively state properties of our decomposition (see Theorem 5.1 in Section 5). Finally, in Section 6, we illustrate our methodology, using existing un-searchable and imbalanced concurrent data structures [13] and [1] as black boxes. Quantitative bounds for resulting (searchable and balanced) distributed data structures are given in Corollary 6.3.

To simplify the presentation, we will assume throughout the paper that sites work reliably and that sites leave gracefully, i.e. if a site s wants to leave the system, its first waits for $s.\mathsf{Leave}()$ to complete before, for example, closing the Internet connection. Certainly, for a distributed data structure implementation to work reliably in a distributed setting, fault and exception handling is a major issue, but considering this here would have made the paper unreadable and would have disguised our main new ideas.

1.2 Previous work

The essence of our paradigm to construct a distributed data structure is to combine two concurrent data structures in a transparent and consistent way. In this section, we give an overview of previous work on concurrent data structures and consistency.

Concurrent data structures

The difference between a sequential and a concurrent data structure is that each data item (resp. the object holding it) in a concurrent data structure can be used as a handle into the data structure and that

*Supported by DARPA grant F306020020550 "A Cost Benefit Approach to Fault Tolerant Communication" and DARPA grant F30602000-2-0526 "High Performance, Robust and Secure Group Communication for Dynamic Coalitions".

the operations Insert, Delete, and Search can be executed concurrently. Concurrent data structures have been heavily investigated in the area of shared memory machines (or PRAMs) [14], though in a slightly weaker form than we require here, since in some of these constructions not every data item can be used as a handle. The reason this was often not necessary is that these approaches rather concentrate on parallelizing the *way* in which the data structure is accessed instead of parallelizing its *structure*, i.e. making its structure more symmetric. For distributed environments, symmetric concurrent data structures are of much greater use because of much better fault tolerance properties.

Symmetric concurrent data structures form the essence of our construction. The problem of managing such a data structure is very similar to maintaining a searchable overlay network between dynamically changing (and reliable) sites with names. In this sense, elegant randomized implementations for concurrent searchable data structures have already been presented, essentially independently, by Li and Plaxton [7], Aspnes and Shah [1] and Harvey et al [5]. (Note that hash-based methods [13, 11, 12, 15, 8] do not work because they only allow precise lookup.) The resulting data structures were named *skip graphs* or *skip nets* [7, 1, 5]. They constitute a simple and elegant extension of the randomized skip list data structure proposed by Pugh [10] to a concurrent environment. In [7, 1, 5] the focus is on proving upper bounds on the degree, path length, and expansion of these constructions.

The reason why the concurrent searchable data structure problem is easier than the distributed variant of this problem is that it leaves *unsolved* how to dynamically embed a data structure on top of the site structure. The contribution of our paper is a simple paradigm accomplishing such an embedding, thus making the results of [7, 1, 5] applicable to distributed, load-balanced message-passing peer-to-peer networks. Next, we describe the approach of [7, 1, 5].

Existing concurrent data structures

Essentially all the solutions for concurrent searchable data structures are based on a doubly-linked list or cycle of data sorted in increasing order of their (user-defined) names. Each data item y has a pointer to its successor $z = \text{succ}(y)$ and predecessor $x = \text{pred}(y)$. 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, 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 o keeps pointers to all data o whose ranking in this cycle is exactly 2^i larger than its own, namely $\mathsf{point}_i(o) = o'$ such that

$$\text{rank}(o') = \text{rank}(o) + 2^i \pmod n$$

where $\text{rank}(o) = r - 1$ if o has the rth smallest name. The pointer graph resulting from this belongs to the class of chordal graphs and is close to a hypercube. Because of the expansion properties of the hypercube, it can be shown that, in a data structure as above with n data items, n concurrent search operations with constant contention at the endpoints create an only logarithmic amount of traffic 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 [10] and extended to parallel environments in [7, 1, 5].

The basis of these data structures is a doubly-linked, sorted list ℓ of all data items. Imagine that we pad the name of each item f with a bit 0 or 1 in the least significant position, and not all the pads are equal. Now, we can decompose ℓ into two sorted sublists: ℓ_0, which contains all sorted even data (i.e. data padded with 0), and ℓ_1, which contains all sorted odd data (i.e. data padded with 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 $\ell_{b_1,b_2\ldots b_i}$ is a doubly-linked list of all data with the padding sequence $b_1, b_2 \ldots b_i$.

The following approaches to padding are possible.

- *Alternate bit padding:* Even-positioned nodes choose 0, and odd positioned nodes choose 1. Then ℓ_0 will contain even-positioned nodes and ℓ_1 will contain odd-positioned nodes. In this case,

  $$\mathsf{point}_i(o) = o' \text{ s.t. } \text{rank}(o') = \text{rank}(o) + 2^i$$

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

- *Random bit padding:* If the numbers chosen by the nodes are random, then on expectation, $\text{rank}(\mathsf{point}_i(o)) - \text{rank}(o) = \Theta(2^i)$, and we get an approximation to a hypercube. The advantage of random choices is that its distributed implementation of an Insert and Delete operation is easy. For example, deleting data item y with predecessor x and successor z in a sublist $\ell_{b_1,b_2\ldots b_i}$ involves simply changing x's (resp. z's) pointer 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 [7, 1, 5].

Consistent Hashing: assignment of files to sites

Next we consider previous work on consistent memory assignment related to our approach. Virtually all distributed data structures presented so far (e.g., [13, 11, 12, 15, 8]) use consistent hashing. Consistent hashing was introduced by Karger et al. [6] 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 or 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: First of all, 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 or random hash functions h and g is that this makes the data structure unsearchable in our sense, since hashing scrambles the name space, making prefix search impossible (i.e. only precise key lookup operations can be supported). On the other hand, replacing hashing by a structure-preserving function g, such as the identity function, to enable searchability may create a high load imbalance. This paper, however, will

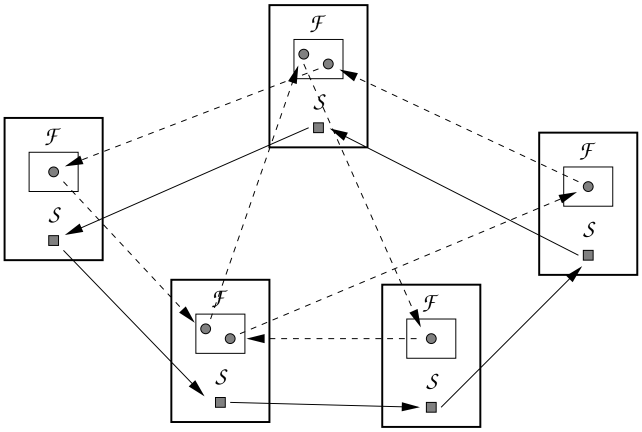

Figure 1: A distributed data structure $\mathcal{D}$ whose file nodes and site nodes form separate topologies representing directed cycles.

get the best of both worlds: functionality (i.e., prefix search capability) of [7, 1, 5] and the load balancing property of [13, 11, 12, 15, 8], thus solving the open problem in [4].

2. THE PLUG-AND-PLAY PARADIGM

In this section, we describe on a high level how to combine concurrent data structures in a transparent and consistent way to construct an efficient, distributed searchable data structure $\mathcal{D}$. Implementation details will be given in the next section. In the following, let Files denote the current set of data items (also called *files* in the following to distinguish between data stored in different data structures) and Sites denote the current set of sites.

2.1 Data structures

Instead of only organizing the sites in a data structure, we use data structures for both the sites and the files (see Figure 1). More precisely, our construction needs the following data structures:

- A concurrent searchable data structure $\mathcal{F}$ for Files. This may be any one suggested in [7, 1, 5].
- A concurrent data structure $\mathcal{S}$ for Sites supporting just lookup operations. This may be any one suggested in [13, 11, 15, 8].

To make sure that both of these data structures are accessible from any site, each site s has exactly one data item in $\mathcal{S}$ representing s and one data item, called *auxiliary handle*, in $\mathcal{F}$ representing s. Apart from these two items, a site may also have many other data items in $\mathcal{F}$ representing files that have been inserted into the system. Each of these data items may potentially be used as a handle into the corresponding structure.

To ensure a plug-and-play property, $\mathcal{F}$ and $\mathcal{S}$ have to be able to operate independently of each other, i.e. changes in Files should *only* affect $\mathcal{F}$, and changes in Sites should *only* affect $\mathcal{S}$ (and $\mathcal{F}$ only insofar that auxiliary handles are inserted or deleted). Next we explain how to achieve this.

2.2 Consistent naming

We use a different name space for each of the data structures.

- $\mathcal{F}$ uses the name space Names representing the set of all possible user-defined names of files (and the space $[0, 1)$ for the auxiliary handles).

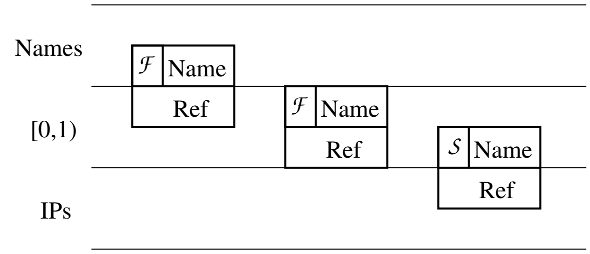

Figure 2: Names and references of data items in $\mathcal{F}$ and $\mathcal{S}$. The middle box represents auxiliary handles into $\mathcal{F}$.

- $\mathcal{S}$ uses the name space $[0, 1)$.
- $\mathcal{D}$ uses the name space IPs consisting of all possible IP addresses.

These name spaces are glued together in a consistent way by assigning to each data item o not only a name $\mathsf{Name}(o)$ but also a reference $\mathsf{Ref}(o)$, which specifies its location. We use the following rules for the references (see also Figure 2):

- Data items in $\mathcal{F}$ have references in the name space $[0, 1)$, and a data item o is stored at the site s with minimum $\mathsf{Name}(s)$ so that $\mathsf{Name}(s) \geq \mathsf{Ref}(o)$.
- Data items in $\mathcal{S}$ have references in the name space IPs, i.e. the reference of a data item in $\mathcal{S}$ is equal to the IP address of its site.

Now suppose that the files in $\mathcal{F}$ choose their references via some random hash function g : Files $\rightarrow [0, 1)$ and that the sites in $\mathcal{S}$ choose their names via some random hash function h : IPs $\rightarrow [0, 1)$. Then we have exactly the same situation as in the consistent hashing approach [6].

The consistent naming approach achieves independence between $\mathcal{F}$ and $\mathcal{S}$ (apart from the auxiliary handles), because any change in Files does not affect the name or reference of a site, and any change in Sites does not affect the name or reference of a file (but only its place).

Apart from gluing name spaces together in a consistent way, including references in data items also has the nice effect that in order for item o to establish a pointer to item o', it just has to store a copy of o' in its neighbor list. By doing this, an item in $\mathcal{S}$ can determine the IP address of its neighbor and forward a request directly to it. An item o in $\mathcal{F}$ can forward a request to its neighbor o' by selecting $\mathsf{Ref}(o') \in [0, 1)$ and then asking $\mathcal{S}$ to forward the request to the site responsible for $\mathsf{Ref}(o')$. Or more precisely, to preserve independence between $\mathcal{F}$ and $\mathcal{S}$, $\mathcal{D}$ will ask $\mathcal{S}$ on $\mathcal{F}$'s behalf to forward the request to the corresponding site. We will give more details about how this works later.

Certainly, it would be easier to forward requests if $\mathsf{Ref}(o)$ of an item o in $\mathcal{F}$ stored the IP address of its site instead of some number in $[0, 1)$. This, however, would have a devastating effect. First of all, it destroys consistency if for some reason a site leaves and not all references of objects affected by this (which includes the neighbors of objects that have to move) are informed about it. Also, a change in Sites could create a tremendous amount of update work, because the number of files in a site can be large, and therefore their neighbors in $\mathcal{F}$ may be spread out among a large number of sites that all have to be updated. We note, however, that for optimization purposes $\mathcal{D}$ may have a cache storing relationships between references of files and the IP addresses of their sites.

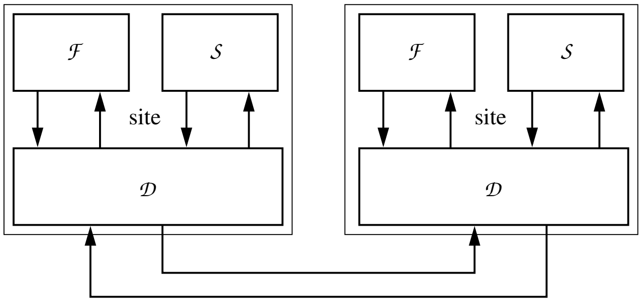

Figure 3: Nodes in $\mathcal{F}$ and $\mathcal{S}$ can only communicate with their local $\mathcal{D}$-node. $\mathcal{D}$-nodes may then pass the messages from $\mathcal{F}$ and $\mathcal{S}$ on to other $\mathcal{D}$-nodes to deliver them to the right recipient. To ensure modularity, there is no direct communication between an $\mathcal{F}$-node and an $\mathcal{S}$-node.

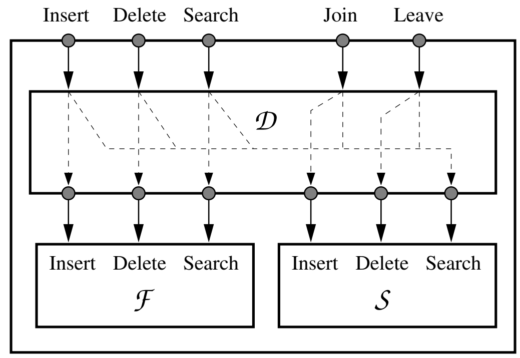

Figure 5: Operations in $\mathcal{D}$ and their connections to operations in $\mathcal{F}$ and $\mathcal{S}$.

2.3 Supporting operations in $\mathcal{F}$ and $\mathcal{S}$

$\mathcal{D}$ supports communication between items in $\mathcal{F}$ and items in $\mathcal{S}$ by essentially providing a transport layer for $\mathcal{F}$ and $\mathcal{S}$. (See Figure 3 to see the communication lines between the data structures and Figure 4 to see how $\mathcal{D}$ would allow two items in $\mathcal{F}$ to communicate.) That is, each time an item o in $\mathcal{S}$ or $\mathcal{F}$ wants to send a message to a neighbor o', $\mathcal{D}$ will take care of that. This ensures that our approach is modular, i.e. any implementation of a concurrent data structure fulfilling the interface requirements specified in the next section can be plugged into $\mathcal{D}$ without further adaptations.

2.4 Operations in $\mathcal{D}$

Suppose that a site s wants to join $\mathcal{D}$. Then this is done by inserting an item representing s into $\mathcal{S}$ and an auxiliary handle representing s into $\mathcal{F}$ to give s a handle into both data structures. Afterwards, all files in $\mathcal{F}$ s is responsible for are moved to s. The insertions executed within the join operation are handled by the corresponding Insert operations in $\mathcal{F}$ and $\mathcal{S}$.

If a site s wants to leave $\mathcal{D}$, then it first moves all of its files in $\mathcal{F}$ to the site now responsible for them, then deletes its auxiliary handle from $\mathcal{F}$, and finally deletes its data item from $\mathcal{S}$. The deletions are handled by the corresponding Delete operations in $\mathcal{F}$ and $\mathcal{S}$.

If a file f has to be inserted into $\mathcal{D}$, then the site initiating the request first contacts the site responsible for f. This site will then include f in its set of files and insert f into $\mathcal{F}$ by calling the corresponding Insert operation.

If a file f has to be deleted from $\mathcal{D}$, then the site initiating the request first contacts the site responsible for f. This site will then delete f from $\mathcal{F}$ by calling the corresponding Delete operation and then remove f from its set of files.

Finally, if a search request for some name key is issued by some site s, then s picks a handle into $\mathcal{F}$ (which it always has due to its auxiliary handle) and calls from there the corresponding Search operation for $\mathcal{F}$.

2.5 Remarks on the P&P paradigm

At the end of this section, we note some nice features of our plug-and-play paradigm that may be interesting for future research on peer-to-peer systems.

Due to explicitly storing references, we do not need hash functions for the data items and the sites to achieve searchability. *Any* way of selecting references for items in $\mathcal{F}$ and names for items in $\mathcal{S}$ would work. Hence, in contrast to previous work, *any* mapping of data items to sites can in principle be supported. This opens up many interesting applications. For example, the freedom of choosing an arbitrary mapping may be used to handle non-uniform capacities, to store popular data items at the sites of highest bandwidth, or to pull data items to the sites that have the largest number of search requests to these items. Hence, our plug-and-play approach may be used for many interesting future optimization problems in the area of peer-to-peer networks.

3. IMPLEMENTATION

We use object oriented programming principles to give a detailed description of how to implement a distributed searchable data structure. For an illustration of how this implementation works, see Figure 6.

The lowest form of object is a data object. Each data object o consists of two fields: a name field $o.\mathsf{Name}$ and a reference field $o.\mathsf{Ref}$. The name field is used for searching purposes, and the reference field specifies the location of the object. The format of the location may depend on the concurrent data structure the object belongs to. If o belongs to $\mathcal{S}$, then $o.\mathsf{Ref}$ is an IP address, and if o belongs to $\mathcal{F}$, then $o.\mathsf{Ref} \in [0, 1)$.

3.1 The class $\mathcal{C}$-node

The class $\mathcal{C}$-node defines the variables and methods used for a concurrent data structure node $\mathcal{C}$ (see Figure 7). Its constructor $\mathcal{C}$-node(name, ref) generates an object containing a data object o with name $o.\mathsf{Name} = \mathsf{name}$ and reference $o.\mathsf{Ref} = \mathsf{ref}$. As required for a concurrent data structure, $\mathcal{C}$-node provides the three standard operations Insert, Delete, and Search. Given a $\mathcal{C}$-node v that is already integrated into $\mathcal{C}$, $v.\mathsf{Insert}(o)$ integrates the $\mathcal{C}$-node w containing data object o into $\mathcal{C}$ by setting up the neighbor sets of w and its neighboring nodes in a suitable way. For example, in order to establish an edge between node u to w, u adds $w.\mathsf{MyData}$ to $u.\mathsf{MyPointers}$ and w adds $u.\mathsf{MyData}$ to $w.\mathsf{MyPointers}$. Operation $v.\mathsf{Delete}(\mathsf{key})$ will exclude the $\mathcal{C}$-node w holding the data object o with $o.\mathsf{Name} = \mathsf{key}$ from $\mathcal{C}$ by setting $w.\mathsf{MyPointers} = \emptyset$ and modifying the neighbor sets of the remaining nodes so that o is not in $\mathcal{C}$ any more. $v.\mathsf{Search}(\mathsf{key})$ returns the data object o in $\mathcal{C}$ with $o.\mathsf{Name}$ being the closest prefix from above to key.

To enable the $\mathcal{C}$-nodes to communicate with each other, we also need functions Read and Write. $\mathsf{Write}(\mathsf{ref}, \mathsf{msg})$ sends the message msg to the node with identifier ref via some output stream, and Read() receives incoming messages via some input stream. A message has the format $(\mathsf{ref}', \mathsf{call}, L)$, where ref' specifies the final recipient, call specifies a command that the recipient is supposed to execute, and L is the parameter list necessary to execute the com-

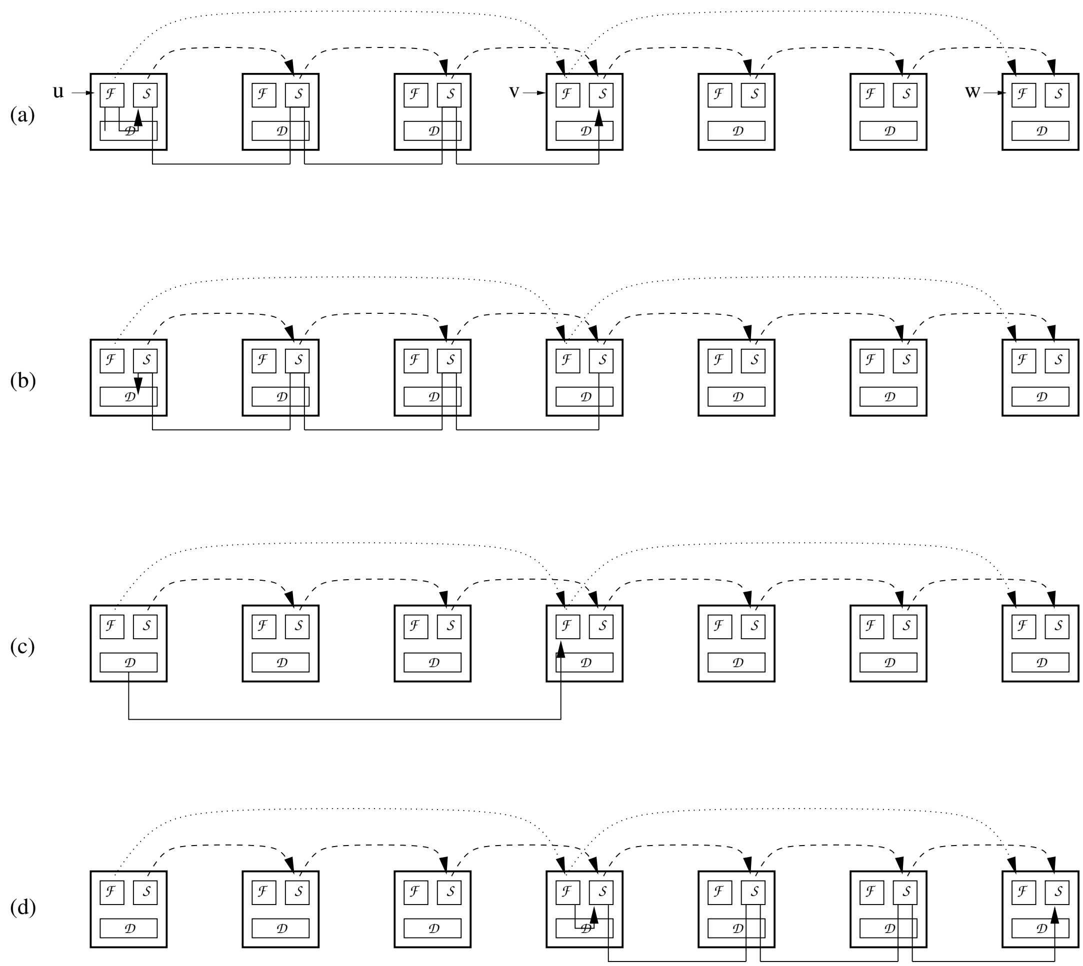

Figure 4: An $\mathcal{F}$-node u wants to forward a message to an $\mathcal{F}$-node w along a path in $\mathcal{F}$ of length two, represented by the dotted arcs. For this, u's $\mathcal{D}$-node d intercepts u's message to v in d.Read() and initiates a search request in $\mathcal{S}$ for Ref(v) (see (a)). The answer of this request is returned by $\mathcal{S}$ (see (b)) and used by d to forward the message from u to v (see (c)). v's message to w is then handled by its $\mathcal{D}$-node d' in the same way as previously for u by d.

mand. Hence, the kind of messages we are interested in are remote procedure calls.

3.2 The class $\mathcal{D}$-node

The class $\mathcal{D}$-node defines the variables and methods used for a distributed data structure node (see Figures 7 and 8). Its constructor $\mathcal{D}$-node(name, ref) generates a $\mathcal{D}$-node d with d.MySite.Name $=$ name and d.MySite.Ref $=$ ref. ref represents, for example, an IP address to allow communication between the $\mathcal{D}$-nodes. The constructor also initializes the set of objects MyFiles representing nodes in the file structure $\mathcal{F}$ to $\emptyset$. As required for a distributed data structure, $\mathcal{D}$ also provides the five standard operations Join, Leave, Insert, Delete, and Search.

3.3 The methods of $\mathcal{D}$-node

If a new node v wants to join $\mathcal{D}$ by contacting w, then it calls v.Join(ref$_w$) where ref$_w$ is the reference (i.e. IP address) identifying w. v.Join(ref$_w$) will then send a message to w asking it to execute LetJoin(o) where $o = v$.MySite. w.LetJoin(o) first searches for the closest successor q of o in $\mathcal{S}$ (i.e. the q with minimum q.MyData.Name so that q.MyData.Name $\geq$ o.MyData.Name). Afterwards, it inserts o into $\mathcal{S}$ (thereby integrating the $\mathcal{S}$-node of v into $\mathcal{S}$) and moves to v all $\mathcal{F}$-nodes stored in the $\mathcal{D}$-node owning q that have to be stored at v to get back to a consistent data placement. Finally, it inserts an auxiliary handle with name and reference equal to v.MySite.Name into $\mathcal{F}$.

Given a node v that wants to leave $\mathcal{D}$, v.Leave() first removes v's auxiliary handle from $\mathcal{F}$ and v.MyFiles, then moves v's $\mathcal{F}$-nodes to the $\mathcal{D}$-node owning the successor of v.MySite in $\mathcal{S}$, and finally removes v's $\mathcal{S}$-node v.MySite from $\mathcal{S}$.

For a $\mathcal{D}$-node v executing v.Insert(o), v first searches for the $\mathcal{S}$-node q with closest name from above to o.Ref by initiating a search operation in $\mathcal{S}$. Afterwards, v asks the $\mathcal{D}$-node w owning q to perform the insertion of o. w checks whether it already has a data object with name o.Name in w.MyFiles. If not, then w will pick a handle f' into $\mathcal{F}$, include a new $\mathcal{F}$-node f holding o into w.MyFiles, and finally include f into $\mathcal{F}$ by calling f'.Insert(o).

The node v initiating v.Delete(key) first checks via a Search operation in $\mathcal{F}$ whether there is a data item with name key in the system. If so, v determines via another search operation in $\mathcal{S}$ the $\mathcal{D}$-node w responsible for key and then asks w to delete the object with name key. w first deletes the data object with name key from $\mathcal{F}$ by

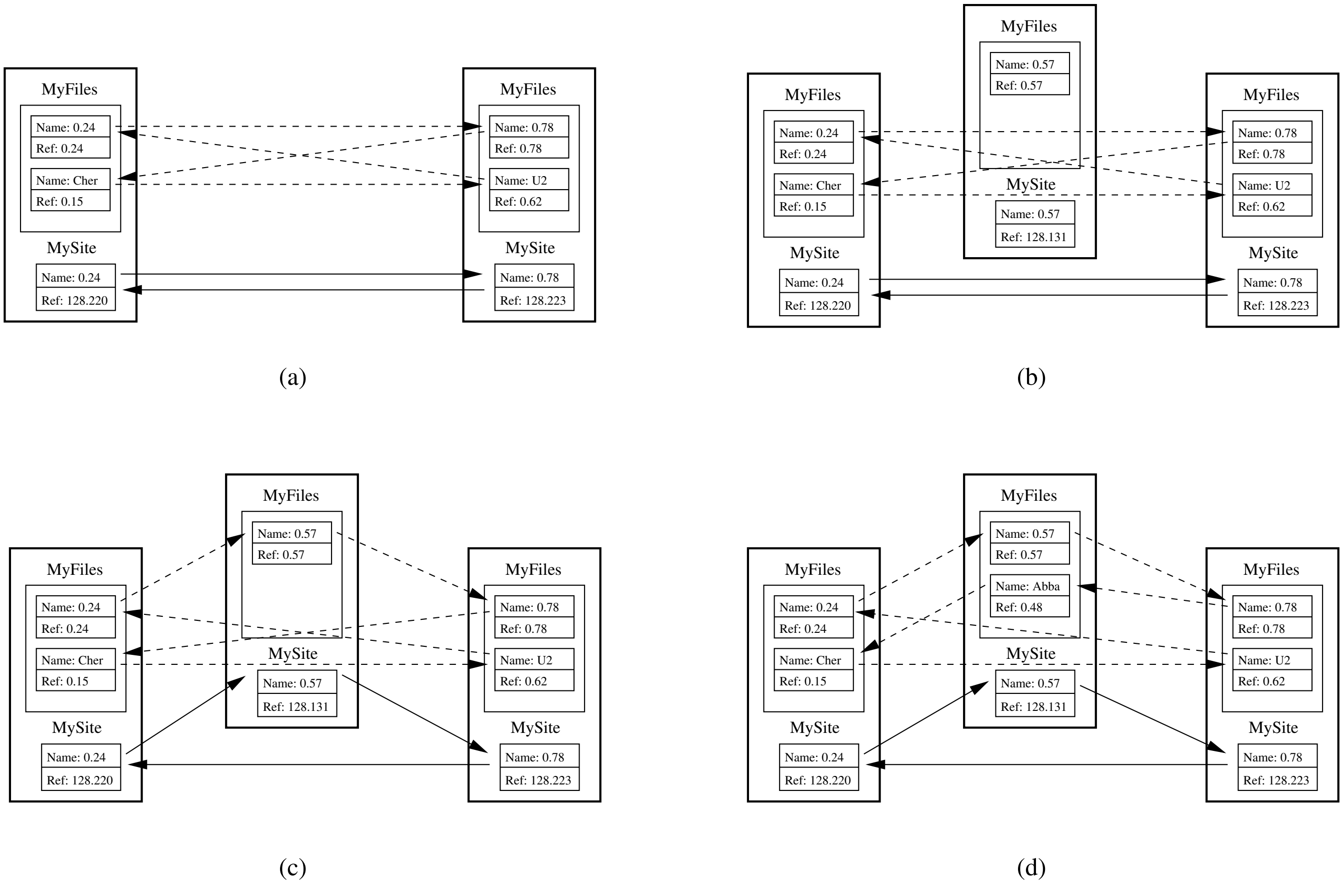

Figure 6: Suppose that $\mathcal{F}$ and $\mathcal{S}$ simply keep the data items in a directed cycle sorted according to their names. The example shows what happens in this case if a $\mathcal{D}$-node joins the system (see (b) and (c)) and the data item "Abba" gets inserted (see (d)).

calling $f.\mathsf{Delete}(\mathsf{key})$ via some handle f into $\mathcal{F}$ and then removes the $\mathcal{F}$-node holding the data item of name key from $w.\mathsf{MyFiles}$.

Finally, to execute $v.\mathsf{Search}(\mathsf{key})$, node v first fetches a handle f into $\mathcal{F}$ and then executes $f.\mathsf{Search}(\mathsf{key})$.

An example code of introducing a new site and a new file is given in Figure 10.

4. PERFORMANCE MEASURES

Next we introduce some measures that will help us to determine the quality of our plug&play approach.

4.1 Sequential data structures

We can view a sequential data structure $\mathcal{U}$ as a directed graph $G_{\mathcal{U}} = (V, E)$ where nodes represent items stored in the data structure and directed edges represent pointers. The *fault-tolerance* $\alpha(G_{\mathcal{U}})$ measures the (lack of) sensitivity to faults, i.e. the ratio between the number of faults and the number of elements disconnected by these faults. This is captured by the expansion of a graph.

DEFINITION 4.1. *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*

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

For a data structure $\mathcal{U}$ and integer n, consider the family $\mathcal{G}_n = \{G(\mathsf{names})\}$ of all worst case pointer structures formed by this data structure over all possible sets $\mathsf{names} \subseteq \mathsf{Names}$ of user-defined names with $|\mathsf{names}| = n$. That is, $G(\mathsf{names})$ represents the worst case pointer graph over all possible sequences of insertion and deletion operations so that at the end, the set of names stored in $\mathcal{U}$ is names.

We will distinguish between metrics for the worst case and the average case for $\mathcal{G}_n$. In order to evaluate the average case, we average over the uniform distribution.

Let the *worst-case* fault-tolerance $\alpha_{\mathcal{U}}(n)$ and the *average-case* fault-tolerance $\bar{\alpha}_{\mathcal{U}}(n)$ of $\mathcal{U}$ be defined as

$$\alpha_{\mathcal{U}}(n) = \min_{G \in \mathcal{G}_n} \alpha(G) \text{ and } \bar{\alpha}_{\mathcal{U}}(n) = \frac{1}{|\mathcal{G}_n|} \sum_{G \in \mathcal{G}_n} \alpha(G)$$

As pointed out in [2], traditional pointer graphs (2-3 trees, linked lists, etc.) are fault-sensitive in the sense that deleting or corrupting a single memory location (i.e., the root of the tree and its pointers) may cause many or even all other locations to become unreachable. Since our ultimate goal is embedding our data structures onto dynamic distributed systems in a fault-tolerant manner, we limit our attention to data structures whose underlying pointer-graph representation is both searchable and fault-tolerant, thus ruling out trees, linked lists, etc with low expansion properties. Fault-tolerant sequential data structures were investigated, e.g., in [2]. We are interested in fault-tolerant concurrent data structures.

```
class Data
{
    Name: name of data item
    Ref: reference to location of data item

public:
    Data(name,ref): constructor setting Name and Ref
}

class C-node
{
    MyData: object of type Data
    MyPointers: set of objects of type Data (neighbors in C)

    Read(): reads message from input stream
    Write(ref, msg): writes msg with dest. ref to output stream

public:
    C(Name,Ref): constructor
    Insert(o): inserts object o into C
    Delete(key): removes object with name key from C
    Search(key): returns data object with closest prefix to key
}

class D-node
{
    MySite: object of type C-node (handle into S)
    MyFiles: set of objects of type C-node (handles into F)

    LetJoin(o): includes site represented by o into D
    OwnerInsert(o): inserts o at site responsible for storing o
    OwnerDelete(key): deletes key from site responsible for key

    GetFile(key): returns pointer to C-node in MyFiles
    InsertFile(f): inserts C-node f into MyFiles
    MoveFiles(o): moves C-nodes in MyFiles to another D-node

    Read(): reads message from input stream
    Write(ref, msg): writes msg with dest. ref to output stream

public:
    D-node(name,ref): constructor
    Join(ref): contacts site in D with address ref to join D
    Leave(): leaves D
    Insert(o): inserts file o into D
    Delete(key): removes file with name key from D
    Search(key): returns file whose name is closest prefix to key
}
```

Figure 7: The classes of $\mathcal{D}$. key and name are of type Names or $[0,1)$, o is of type Data, and f is of type $\mathcal{C}$-node.

4.2 Concurrent data structures

The additional issue in this case is that many insert, delete, or search requests may be traversing the data structure concurrently, and we need to make sure that nodal congestion is not slowing down the process. In order to measure this, we introduce the *searchability* $\rho(G)$.

Consider a concurrent data structure $\mathcal{C}$ consisting of

- a pointer structure G and
- a probabilistic or deterministic implementation of Search.

A *random search problem* in a pointer structure G is a routing problem in which each node in G is the source of a request and each request chooses its destination independently at random. Looking at random destinations is useful for peer-to-peer systems that (pseudo-)randomly distribute data among the sites, which includes all distributed hash table approaches [13, 11, 15, 8].

DEFINITION 4.2. *Given a random search problem in a pointer structure G, let*

- *$C(G)$ be the expected congestion (i.e. the expected maximum number of paths sharing a node) and*
- *$D(G)$ be the dilation (i.e. the length of the longest path) when applying* Search *to the requests in G.*

Then the searchability *of G is $\rho(G) = \max\{C(G),\ D(G)\}$.*

The following claim gives an important relationship between the searchability and the expansion.

CLAIM 4.3. *For every graph G with searchability $\rho(G)$, $\alpha(G) = \Omega(1/\rho(G))$.*

PROOF. Suppose that there is a set $U \subseteq V$ with $|U| \leq |V|/2$ and $|\Gamma(U)| = o(|U|/\rho(G))$. Then consider the search problem in Definition 4.2. It is easy to see that the expected number of requests that have to leave or enter U is equal to

$$2|U|\left(1 - \frac{|U|}{|V|}\right) \geq |U|$$

for all U with $|U| \leq |V|/2$. Hence, the expected congestion created at nodes in $\Gamma(U)$ due to requests leaving or entering U is at least $|U|/|\Gamma(U)|$. On the other hand, it follows from the definition of searchability that the expected maximum congestion is at most $\rho(G)$. However, this is not possible to achieve if $|\Gamma(U)| = o(|U|/\rho(G))$. $\square$

Now, consider the family of worst case pointer structures $\mathcal{G}_n = \{G(\text{names})\}$ formed by the given data structure $\mathcal{C}$ over all possible sets names $\subseteq$ Names of user-defined names of size n. Let the *worst-case* searchability $\rho_{\mathcal{C}}$ and the *average-case* searchability $\bar{\rho}_{\mathcal{C}}$ of $\mathcal{C}$ be defined as

$$\rho_{\mathcal{C}}(n) = \max_{G \in \mathcal{G}_n} \rho(G) \ \text{ and } \ \bar{\rho}_{\mathcal{C}}(n) = \frac{1}{|\mathcal{G}_n|} \sum_{G \in \mathcal{G}_n} \rho(G)$$

4.3 Distributed data structures

In a distributed system, we need not only to create a data structure but also to embed it on a set of sites. To take this into account, we suggest the following fault-tolerance measure for distributed data structures.

DEFINITION 4.4. *Consider an instance D of a distributed data structure over a set of files* Files *and a set of sites* Sites *with a search operation* Search *for the files. For a set of files $F \subset$* Files *and a set of sites $S \subset$* Sites *we say that S* blocks *F, denoted as $S \oslash F$, if every* Search(Name(f)) *operation with $f \in F$ executed by a*

```
d.D-node(name, ref):
    // prepare handle for S
    MySite ← new C(name, ref)
    // prepare auxiliary handle for F
    f ← new C(name, name)
    MyFiles ← {f}

d.Join(ref):
    Write(ref,(∅,call_LetJoin,MySite))

d.LetJoin(o):
    q ← MySite.Search(o.Name)
    // integrate site node of o into S
    MySite.Insert(o)
    Write(q.Ref,(∅,call_MoveFiles,o))
    // integrate auxiliary file node into F
    o.Ref ← o.Name
    f ← GetFile(o.Name)
    f.Insert(o)

d.MoveFiles(o):
    for all f ∈ MyFiles with f.Ref ≤ o.Name
    Write(o.Ref,(∅,call_InsertFile,f))

d.InsertFile(f):
    MyFiles ← MyFiles ∪ {f}

d.Leave():
    // remove auxiliary file node from F
    f ← GetFile(MySite.Name)
    f.Delete(MySite.Name)
    // move file nodes to predecessor of MySite in S
    q ← MySite.Search(MySite.Name + ϵ)
    MoveFiles(q)
    // remove site node from S
    MySite.Delete(MySite.Name)

d.GetFile(key):
    if there is f ∈ MyFiles with f.MyData.Name = key then
        return f
    else
        return a random f ∈ MyFiles

d.Insert(o):
    // determine site responsible for o
    q ← MySite.Search(o.Ref)
    Write(q.Ref,(∅,call_OwnerInsert,o))

d.OwnerInsert(o):
    // insert o into F via f′
    f′ ← GetFile(o.Name)
    if f′.MyData.Name ≠ o.Name then
        f ← new C-node(o.Name, o.Ref)
        MyFiles ← MyFiles ∪ {f}
        f′.Insert(o)

d.Delete(key):
    // is key in the system?
    o ← Search(key)
    if o.Name = key then
        // determine site responsible for o
        q ← MySite.Search(o.Ref)
        Write(q.Ref,(∅,call_OwnerDelete,key))

d.OwnerDelete(key):
    // remove file object with name key from F
    f ← GetFile(key)
    f.Delete(key)
    MyFiles = MyFiles \ {f : f.MyData.Name = key}

d.Search(key):
    f ← GetFile(key)
    return f.Search(key)
```

Figure 8: The methods of $\mathcal{D}$. d represents a $\mathcal{D}$-node.

site in Sites $\setminus S$ *passes through at least one site in* S*. Denote the fault-tolerance of the resulting data structure as*

$$\alpha(D) = \mathrm{E}\left[\min_{S \oslash F} \frac{|S|/|\mathsf{Sites}|}{|F|/|\mathsf{Files}|}\right]$$

where the expected value is over the mappings of Files *to* Sites.

Intuitively, $\alpha(D)$ shows how much percentage of the sites need to be down to block access to one percentage of the files.

Next, we define the imbalance and searchability of a distributed data structure.

DEFINITION 4.5. *Given a mapping of files to sites, let $L(s)$ be the load (i.e. the number of files) at site s. The* imbalance $\sigma_{\mathcal{D}}(k)$ *of a distributed data structure $\mathcal{D}$ is the maximum over all sets of files* Files *and sets of sites* Sites *with* $|\mathsf{Files}|/|\mathsf{Sites}| = k$ *of*

$$\mathrm{E}\left[\max_{s \in \mathsf{Sites}} \frac{L(s)}{|\mathsf{Files}|/|\mathsf{Sites}|}\right]$$

where the expected value is over the mappings of Files *to* Sites.

DEFINITION 4.6. *We define the* searchability $\rho(D)$ *of instance D as the maximum of the dilation and expected congestion caused by sending search requests, one for each site, to random files.*

Now, consider the family of structures $\mathcal{G}_{n,m} = \{D(\mathsf{names}, \mathsf{IPs})\}$ formed by the distributed data structure $\mathcal{D}$ over all possible instances names of file names and IPs of IP addresses of sites, $|\mathsf{names}| = m$ and $|\mathsf{IPs}| = n$. Then we define the worst-case and average-case fault-tolerance and searchability of $\mathcal{D}$ as

$$\alpha_{\mathcal{D}}(n,m) = \min_{D \in \mathcal{G}_{n,m}} \alpha(D) \text{ and } \bar{\alpha}_{\mathcal{D}}(n,m) = \frac{1}{|\mathcal{G}_{n,m}|} \sum_{D \in \mathcal{G}_{n,m}} \alpha(D)$$

and

$$\rho_{\mathcal{D}}(n,m) = \max_{D \in \mathcal{G}_{n,m}} \rho(D) \text{ and } \bar{\rho}_{\mathcal{D}}(n,m) = \frac{1}{|\mathcal{G}_{n,m}|} \sum_{D \in \mathcal{G}_{n,m}} \rho(D)$$

From Definitions 4.5 and 4.4, and 4.6 it follows that $\alpha_{\mathcal{D}}^{-1} \geq \sigma_{\mathcal{D}}$ and $\rho \geq \sigma_{\mathcal{D}}$ for any m and n.

5. COMPOSITION THEOREM

```
d.Write(ref′,(ref,call,L)):
    send (ref,call,L) to output stream ref′

d.Read():
    upon receiving message (ref,call,L) from ref′, check:
    if ref′ ∈ [0,1) then
        // from F, send msg to site q responsible for ref
        q ← MySite.Search(ref)
        Write(q.Ref,(ref,call,L))
    else if ref′ ∈ IPs then
        if ref ∈ IPs then
            // from S-node
            Write(ref,(ref,call,L))
        else
            // remote procedure call from other D-node
            case call of
                call_LetJoin: LetJoin(L)
                call_MoveFiles: MoveFiles(L)
                call_InsertFile: InsertFile(L)
                call_OwnerInsert: OwnerInsert(L)
                call_OwnerDelete: OwnerDelete(L)
```

Figure 9: **The Write and Read algorithm of $\mathcal{D}$.**

```
MyExample:
    // create new site
    ref ← 120.128.220.114    // IP address of site
    val ← h(ref)    // hash value of IP address
    d ← new D-node(val, ref)
    // join D
    ref ← 120.128.220.135    // IP address of site in D
    d.Join(ref)

    // insert file
    name ← "Hallo"
    val ← g(name)    // hash value of name
    o ← new Data(name, val)
    d.Insert(o)

    // leave D
    d.Leave()
```

Figure 10: **Example code demonstrating how to join $\mathcal{D}$, insert a file, and leave $\mathcal{D}$. h and g are some suitable hash functions. Notice that the values for val can be *arbitrary* numbers in $[0, 1)$, i.e. it is not necessary to use fixed functions for $\mathcal{D}$ to be consistent and searchable.**

In this section we state our main technical result of the paper. Consider a distributed data structure obtained by using $\mathcal{F}$ as a file structure and $\mathcal{S}$ as the site structure, within the framework of Sections 2 and 3. We will denote this as $\mathcal{D} = \mathcal{F} \times \mathcal{S}$.

THEOREM 5.1 (COMPOSITION THEOREM). *If $m = |\mathsf{Files}| \geq |\mathsf{Sites}| = n$, files are given random references in $[0, 1)$, and sites are given random names in $[0, 1)$, then $\mathcal{D} = \mathcal{F} \times \mathcal{S}$ has, in the worst case,*

$$\sigma_{\mathcal{D}} = \Theta(\log n) \quad (1)$$

$$\rho_{\mathcal{D}} = O(\sigma_{\mathcal{D}}^2 \cdot \rho_{\mathcal{F}} \cdot \bar{\rho}_{\mathcal{S}}) \quad (2)$$

$$\alpha_{\mathcal{D}} = \Omega\left(\frac{1}{\sigma_{\mathcal{D}} \cdot (1 + \frac{1}{\bar{\alpha}_{\mathcal{S}}})(1 + \frac{1}{\alpha_{\mathcal{F}}})}\right) \quad (3)$$

PROOF. The proof of (1) follows from the fact that the maximum range in $[0, 1)$ a site is responsible for is $\Theta((\log n)/n)$ with high probability.

We proceed with the proof of (2). Consider the problem of routing a random search problem in $\mathcal{D}$. Then each $\mathcal{F}$-node is the origin of at most one request and each source-destination pair (f, f') in $\mathcal{F}$ has a probability of at most $1/|\mathsf{Files}|$ of being chosen. Since the total number of nodes in $\mathcal{F}$, $n_{\mathcal{F}}$, is at most $2 \cdot |\mathsf{Files}|$ because of at most $|\mathsf{Files}|$ auxiliary handles, each pair (f, f') in $\mathcal{F}$ has a probability of at most $2/n_{\mathcal{F}}$ of being chosen. Hence, it follows from the definition of ρ that routing the requests in $\mathcal{F}$ creates an expected congestion of at most $2\rho_{\mathcal{F}}$ and a dilation (path length) of at most $\rho_{\mathcal{F}}$. Let $\mathcal{P}$ be the path collection chosen by $\mathcal{D}$, and let C be the congestion and D be the dilation of $\mathcal{P}$. We know that $D \leq \rho_{\mathcal{F}}$ and $\mathrm{E}[C] \leq 2\rho_{\mathcal{F}}$. To continue, we need the following two lemmata. Recall that a matching in a graph G is a set of edges so that every node in G is adjacent to at most one edge.

LEMMA 5.2. *$\mathcal{P}$ can be broken down into $O(C + D)$ matchings of size at most $|\mathsf{Sites}|$ each.*

PROOF. The matching bound follows from a simple coloring argument. Since each edge shares a node with an expected number of at most $2C - 1$ other edges, an expected number of at most $2C$ colors suffice to color the edges so that no two edges with the same color share a node. (Simply use a greedy coloring algorithm for this.) Since the sum of the path lengths in $\mathcal{P}$ is at most $|\mathsf{Sites}| \cdot D$, it follows that these matchings can be further broken down into at most D additional matchings to obtain matchings of maximum size $|\mathsf{Sites}|$. □

LEMMA 5.3. *Any matching in $\mathcal{F}$ of size at most $|\mathsf{Sites}|$ can be routed in $\mathcal{S}$ with dilation at most $\bar{\rho}_{\mathcal{S}}$ and expected congestion at most $\sigma_{\mathcal{D}}^2 \cdot \bar{\rho}_{\mathcal{S}}$.*

PROOF. The consistent naming approach and the definition of $\sigma_{\mathcal{D}}$ imply that the probability of a file to be mapped to a site is at most $\sigma_{\mathcal{D}}/n$ (where $n = |\mathsf{Sites}|$) for any site, independently of the other files. Since we only consider routing matchings M in $\mathcal{F}$ in $\mathcal{S}$, each edge in M is mapped independently of the other edges to sites in $\mathcal{S}$. Viewing the probabilities of mapping edges to site pairs as fractional flow values, any site pair (s_1, s_2) has a fractional flow of at most $n \cdot (\sigma_{\mathcal{D}}/n)^2 = \sigma_{\mathcal{D}}^2/n$. From the definition of $\bar{\rho}_{\mathcal{S}}$ we know that fractional flow problems in $\mathcal{S}$ with flow $1/n$ for each site pair can be solved with congestion and dilation at most $\bar{\rho}_{\mathcal{S}}$. Hence, routing the fractional flow problem for M in $\mathcal{S}$ creates a congestion of at most $\sigma_{\mathcal{D}}^2\bar{\rho}_{\mathcal{S}}$, and therefore routing M in $\mathcal{S}$ creates an expected congestion of at most $\sigma_{\mathcal{D}}^2\bar{\rho}_{\mathcal{S}}$. Since the dilation is obviously bounded by $\bar{\rho}_{\mathcal{S}}$, the lemma follows. □

Combining the two lemmata finishes the proof of (2)

Next we prove (3). Consider some instance D with fixed imbalance $\sigma(D)$, $\alpha_{\mathcal{F}}(D)$, and $\alpha_{\mathcal{S}}(D)$. Let $\Lambda_{\mathcal{S}}$ be the set of sites removed, and $\Gamma_{\mathcal{S}}$ be the set of sites in $\mathcal{S}$ inaccessible in $\mathcal{S}$ from a majority in $\mathcal{S}$ that are not in $\Lambda_{\mathcal{S}}$. Furthermore, let $\Upsilon_{\mathcal{S}}$ be the set of sites that can reach the majority in $\mathcal{S}$. For a subset F (S) of files (sites) denote by $||F||$ ($||S||$) the relative fraction of such files (sites) i.e., ratio $|F|/|\mathsf{Files}_D|$ (respectively, $|S|/|\mathsf{Sites}_D|$).

Since $\mathcal{S}$ has a fault-tolerance of $\alpha_{\mathcal{S}}(D)$,

$$||\Gamma_{\mathcal{S}}|| \leq \frac{||\Lambda_{\mathcal{S}}||}{\alpha_{\mathcal{S}}(D)} .$$

Now, let $\Lambda_{\mathcal{F}}$ be the set of files stored at $\Lambda_{\mathcal{S}}$ and $\Gamma_{\mathcal{F}}$ be the *maximal* set of files that are inaccessible in $\mathcal{D}$ from a majority in $\mathcal{S}$, which are not in $\Lambda_{\mathcal{F}}$. Also, let $\Upsilon_{\mathcal{F}}$ be the set of those files which are accessible in $\mathcal{D}$ from majority in $\mathcal{S}$. Note that the sets $\Lambda_{\mathcal{F}}$, $\Upsilon_{\mathcal{F}}$, and $\Gamma_{\mathcal{F}}$ are disjoint in $\mathcal{F}$.

Now, let us decompose $\Gamma_{\mathcal{F}} = \Gamma^1_{\mathcal{F}} \bigcup \Gamma^2_{\mathcal{F}}$ such that $\Gamma^1_{\mathcal{F}} \bigcap \Gamma^2_{\mathcal{F}} = \emptyset$ and $\Gamma^1_{\mathcal{F}}$ is a maximal subset stored at sites in $\Gamma_{\mathcal{S}}$. By the above argument, the fraction of such files is at most

$$||\Gamma^1_{\mathcal{F}}|| \leq ||\Gamma_{\mathcal{S}}|| \cdot \sigma(D) \leq \frac{||\Lambda_{\mathcal{S}}||}{\alpha_{\mathcal{S}}(D)} \cdot \sigma(D)$$

Now, consider file $o \in \Gamma^2_{\mathcal{F}}$ stored at site s. By definition, $s \notin \Lambda_{\mathcal{S}}$ (since then $o \in \Lambda_{\mathcal{F}}$) and also $s \notin \Gamma_{\mathcal{S}}$ since then we could have added o to $\Gamma^1_{\mathcal{F}}$, but $\Gamma^1_{\mathcal{F}}$ cannot be increased since it is maximal.

The only remaining possibility is that $s \in \Upsilon_{\mathcal{S}}$. Consider any file o' which is neighbor of o in $\mathcal{F}$ stored at s'. Clearly, $s' \notin \Upsilon_{\mathcal{S}}$ since then both s and s' are in $\Upsilon_{\mathcal{S}}$, and then o must be in $\Upsilon_{\mathcal{F}}$, a contradiction. It follows that $s' \in \Gamma_{\mathcal{S}} \cup \Lambda_{\mathcal{S}}$. The fraction of such o' is at most

$$\sigma(D) \cdot (||\Lambda_{\mathcal{S}}|| + ||\Gamma_{\mathcal{S}}||) \leq \sigma(D) \cdot \left(1 + \frac{1}{\alpha_{\mathcal{S}}(D)}\right) ||\Lambda_{\mathcal{S}}||$$

Since $o \in \Gamma^2_{\mathcal{F}}$ has all its neighbors in $\mathcal{F}$ in a set of size $(1 + \frac{1}{\alpha_{\mathcal{S}}(D)})||\Lambda_{\mathcal{S}}||$ at most, we have

$$||\Gamma^2_{\mathcal{F}}|| \leq \frac{\sigma(D)}{\alpha_{\mathcal{F}}(D)} \left(1 + \frac{1}{\alpha_{\mathcal{S}}(D)}\right) ||\Lambda_{\mathcal{S}}||$$

Summing it all up, $||\Gamma_{\mathcal{F}}||$ is equal to

$$\begin{aligned}
& ||\Gamma^1_{\mathcal{F}}|| + ||\Gamma^2_{\mathcal{F}}|| \\
\leq \; & \sigma(D) \cdot \frac{||\Lambda_{\mathcal{S}}||}{\alpha_{\mathcal{S}}(D)} + \frac{\sigma(D)}{\alpha_{\mathcal{F}}(D)} \left(1 + \frac{1}{\alpha_{\mathcal{S}}(D)}\right) ||\Lambda_{\mathcal{S}}|| \\
= \; & \sigma(D) \cdot ||\Lambda_{\mathcal{S}}|| \cdot \left(\frac{1}{\alpha_{\mathcal{F}}(D)} + \frac{1}{\alpha_{\mathcal{S}}(D)} + \frac{1}{\alpha_{\mathcal{F}}(D) \cdot \alpha_{\mathcal{S}}(D)}\right)
\end{aligned}$$

Using the fact that $||\Lambda_{\mathcal{F}}|| \leq \sigma(D) \cdot ||\Lambda_{\mathcal{S}}||$, it follows that $\alpha(D)$ is equal to

$$\frac{||\Lambda_{\mathcal{S}}||}{||\Gamma_{\mathcal{F}}|| + ||\Lambda_{\mathcal{F}}||} \geq \frac{1}{\sigma(D)(1 + \frac{1}{\alpha_{\mathcal{F}}(D)})(1 + \frac{1}{\alpha_{\mathcal{S}}(D)})}$$

Hence, with $1/\sigma(D) = \Theta(1/\sigma_{\mathcal{D}})$ w.h.p. and $\alpha_{\mathcal{F}}(D) \geq \alpha_{\mathcal{F}}$ one can compute a bound for the expected $\alpha(D)$ over random values in $[0, 1)$ for files and sites that matches the bound for $\alpha_{\mathcal{D}}$ in the theorem. □

6. CHOOSING THE COMPONENTS

Now, we show how to use our composition methodology from Sections 2 and 3. The only issue is *how to make proper selections* for the distributed data structure and the final peer-to-peer system, out of concurrent data structures presented in Section 1.2. Towards this goal, we simply need to apply the framework of Section 5 and the results of Theorem 5.1, calculating searchability and fault-tolerance of the composed system by simple substitution into the result of Theorem 5.1.

We exemplify the plug-and-play approach of Theorem 5.1 by choosing Chord [13] as the site structure $\mathcal{S}$ and Skip Graph [1] as the file structure $\mathcal{F}$. Other structures, e.g. Chord++ [3] or [9], can be used to get better bounds on the parameters $\sigma_{\mathcal{D}}$, $\alpha_{\mathcal{D}}$ and $\rho_{\mathcal{D}}$ of the composed scheme. However, for the purpose of clarity and completeness, we restrict ourselves to using already published methods, such as Chord [13], and Skip Graph [1].

CLAIM 6.1 ([3]). *When used for site structure $\mathcal{S}$, Chord with the routing strategy in [3] satisfies $\bar{\rho}_{\mathcal{S}}(Ch) = O(\log^2 n)$ and $\bar{\alpha}_{\mathcal{S}}(Ch) = \Omega(\frac{1}{\log n})$.*

CLAIM 6.2 ([1]). *When used for file structure $\mathcal{F}$, Skip Graph [1] satisfies $\rho_{\mathcal{F}}(Sk) = O(\log^2 n)$ (conjectured) and $\alpha_{\mathcal{F}}(Sk) = \Omega(\frac{1}{\log n})$ (shown in [1]).*

Applying Theorem 5.1, we can compose a distributed data structure SkipChord (abbreviated SkCh) consisting of Skip Graph for files, and Chord for sites.

COROLLARY 6.3. *SkipChord is a load-balanced prefix-search capable peer-to-peer system with the following characteristics:*

$$\begin{aligned}
\sigma_{\mathcal{D}}(SkCh) &= O(\log n) \\
\rho_{\mathcal{D}}(SkCh) &= O(\log^6 n) \\
\alpha_{\mathcal{D}}(SkCh) &= \Omega(\frac{1}{\log^3 n})
\end{aligned}$$

7. REFERENCES

[1] J. Aspnes and G. Shah. Skip graphs. In *SODA 2003*.

[2] Y. Aumann and M.A. Bender. Fault tolerant data structures. In *FOCS 1996*, pages 580–589.

[3] B. Awerbuch and C. Scheideler. Chord++: A congestion-free fault-tolerant concurrent data structure. Unpublished manuscript, 2003. See `http://www.cs.jhu.edu/~scheideler`.

[4] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *IPTPS 2002*.

[5] 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*.

[6] 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 *STOC 1997*, pages 654–663.

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

[8] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC 2002*.

[9] M. Naor and U. Wieder. Novel architectures for P2P applications: The continuous-discrete approach. In *SPAA 2003*.

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

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

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

[13] 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*.

[14] U. Vishkin, W. J. Paul, and H. Wagener. Parallel dictionaries on 2-3 trees. In *ICALP 1983*, pages 597–609.

[15] B.Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. In *UCB Technical Report UCB/CSD-01-1141*, 2001.