12 Distributed data management III

In this section we study distributed data (or file) management methods based on pointer structures and load balancing, and we demonstrate how to use them in peer-to-peer systems.

If we ignore for the moment the problem of handling arrivals and departures of peers, the problem of managing data or files boils down to the problem of maintaining a concurrent searchable data structure. Sequential data structures for searchability are usually implemented as directed labelled pointer graphs, e.g. binary search trees with proper labels ensuring that the cost of the search is proportional to the diameter of the data structure. For that reason, most of the work on data structures focuses on how to keep the diameter of the underlying pointer graph as small as possible, i.e. by balancing trees as in 2-3 or red-black trees.

However, such 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 highly fault-tolerant manner, we limit our attention to data structures whose underlying pointer graph has a high expansion, such as the Hyperring presented in the previous section. We start with some general specifications a searchable data structure has to fulfill.

12.1 Input/Output specs

Sequential Searchable Data Structure. $\mathcal{S}$ is a sequential searchable data structure if it offers the following operations for objects from the universe $\mathsf{All\_Objects}$:

- $\mathsf{Insert}(\mathsf{Name}(o), \mathsf{Ref}(o))$: adds object o with name $\mathsf{Name}(o)$ and reference $\mathsf{Ref}(o)$ to $\mathcal{S}$.
- $\mathsf{Delete}(\mathsf{Name}(o))$: removes o from $\mathcal{S}$.
- $\mathsf{Search}(\mathsf{Name})$ retrieves a reference to the object o^* out of $\mathcal{S}$'s current object set $\mathsf{Objects}$ with largest name smaller or equal to Name, i.e.

$$o^* = \mathsf{argmax}\{\mathsf{Name}(o') \mid o' \in \mathsf{Objects}, \mathsf{Name}(o) \leq \mathsf{Name}\}$$

(Alternatively, one may also formulate $\mathsf{Search}()$ for finding the successor for a name.)

One may also define the "exact" $\mathsf{Lookup}(\mathsf{Name})$ command, which only needs to retrieve a reference to an object if its name is *precisely* specified; this command used in [5, 4, 6, 3] is much easier to implement.

Concurrent Searchable Data Structure $\mathcal{C}$ allows to perform the operations above concurrently.

Distributed Searchable Data Structure $\mathcal{D}$ must implement the above operation on a set of processors $\mathsf{All\_Sites}$ which can join and leave the system, such as connecting and disconnecting from the Internet. As in all other work in the peer-to-peer area, we assume that some external mechanism enables a joining peer to find a peer already in the system. This dynamic collection of peers $\mathsf{Sites} \in \mathsf{All\_Sites}$ is defined by the following operations

- $\mathsf{Join}(\mathsf{Ref}(p))$: peer p joins Sites by passing a reference (e.g., its IP address) to another $p' \in \mathsf{Sites}$.
- $\mathsf{Leave}(p)$: peer p leaves Sites. This removes all the data stored by p.

Any peer (or site) $p \in \mathsf{Sites}$ can invoke any of the operations of section 12.1.

A *Distributed Name Service* is a special case of IP addresses directory, namely tuples of the form $(\mathsf{Name}(p), \mathsf{IP}(p))$ for various sites p, whose names are externally imposed(i.e. host.cs.jhu.edu).

A distributed implementation of a searchable data structure consists of a searchable pointer graph on the set of objects, plus a "name-consistent" assignment π mapping objects into sites that store them. Formally,

Definition 12.1 *An assignment π is* name-consistent *if for all $f \in \mathsf{Objects}$,*

$$\mathsf{Search}(\mathsf{Ref}(f)) = \mathsf{Search}(\mathsf{Name}(\pi(f)).$$

Intuitively, π is *name-consistent* if every object f is assigned to a site $p = \pi(f)$ whose reference is the closest match to the name of f. Note that searching for a file f by its reference, or searching for the site $\pi(f)$ by its name is equivalent, and will return the pointer to site p, namely $\mathsf{Search}(\mathsf{Name}(p)) = \mathsf{Ref}(p)$.

Suppose that we do not use an extra pointer structure for organizing the data. Chord, for example, does not have an extra pointer structure but uses name consistency to find data. There are a number of options for selecting a reference, but neither one seems to work without problems.

Original names: $\mathsf{Ref}(f) = \mathsf{Name}(f)$ makes range queries possible, and in fact achieves logarithmic cost for random names. For worst-case names, this method has a poor performance without rebalancing the data among the sites once in a while.

Completely random names: $\mathsf{Ref}(f) = \mathsf{Random}(f)$ solves the problem of imbalance; now the distribution of objects is perfect, but exhaustive $\Omega(n)$ cost search is necessary.

Hashed names: $\mathsf{Ref}(f) = H(\mathsf{Name}(f))$ where H is a hash function, is suggested in [5, 4, 6, 2] to balance between original and random names. However,

- data structures become *un-searchable*, just like hashing based sequential data structures. They can only be used for implementing $\mathsf{Lookup}(\mathsf{Name})$ with Name being a precisely defined string. However, an imprecise query or range queries are un-implementable.
- hash functions such as SHA-1 are *non-random* by definition, do not provide adequate protection against adversarial input, since properly selected input strings can cause $\Omega(n)$ imbalance, even for precise name queries $\mathsf{Lookup}(\mathsf{Name})$. In fact, even without inverting SHA-1 function, an adversary can easily create arbitrary imbalance in the site load by simply inserting lots of objects f with approximately the same value of $H(\mathsf{Name}(f))$.

There are two solutions out of this dilemma to obtain an efficient, searchable distributed data structure:

- Using the real names of the data object together with a balancing mechanism to repair any imbalance of the data distribution caused by removing or inserting data or sites.

- Organizing the data objects in a searchable pointer structure and mapping the pointer structure *randomly* to the sites.

We start with the latter approach.

12.2 Organizing data in a pointer structure

This approach consists of three components:

- A searchable distributed data structure $\mathcal{P}$ whose objects represent the current set of sites in the system, where the name of each object is set to a number chosen uniformly and independently at random from $[0, 1)$.
- A searchable concurrent data structure $\mathcal{F}$ whose objects represent the current set of files in the system, where each object o representing file f has the property that $\mathsf{Name}(o) = \mathsf{Name}(f)$ (i.e. o carries the original name of f) and $\mathsf{Ref}(o)$ is set to a number chosen uniformly and independently at random from $[0, 1)$, which is used as a virtual memory location for o.
- A consistent (and robust) mapping of $\mathcal{F}$ to $\mathcal{P}$. I.e. object o in $\mathcal{F}$ is stored at the site p with $\mathsf{Search}_{\mathcal{P}}(\mathsf{Ref}(o)) = \mathsf{Ref}(p)$ (and copies of it are stored in the nearest neighbors of p in $\mathcal{P}$ in the case that fault-tolerance has to be provided).

To be precise, $\mathcal{F}$ will also contain an object for every site, with name and reference equal to the random name of the site so that each of the sites has access to $\mathcal{F}$. We assume that the name of a site is smaller (or larger) than every possible file name so that site and file names are not interleaved in $\mathcal{F}$, which would otherwise give a wrong result for a site request $\mathsf{Search}(\mathsf{Name})$.

Because we use *random* values to perform the mapping of the files to the sites, we achieve a *complete decoupling* between sites and files. Next we describe on a high level how to implement operations for $\mathcal{P}$ and $\mathcal{F}$.

Operations for the site structure $\mathcal{P}$

Operations for the site structure have to be executed in the following way:

Join$_{\mathcal{P}}(\mathsf{Ref}(p))$**.** Suppose that p contacted site q to join the system. q then chooses a random $\mathsf{Name}(p)$ $\in [0, 1)$ and executes $\mathsf{Insert}_{\mathcal{P}}(\mathsf{Name}(p), \mathsf{Ref}(p))$.

Leave$_{\mathcal{P}}(p)$**.** We assume here that sufficiently redundant information is available about p so that another site, say q, can immediately take over the role of p if p suddenly leaves. First, q moves objects to achieve a consistent mapping without p and then it calls $\mathsf{Delete}_{\mathcal{F}}(\mathsf{Name}(p))$ to remove p from $\mathcal{F}$ and $\mathsf{Delete}_{\mathcal{P}}(\mathsf{Name}(p))$ to remove p from $\mathcal{P}$.

Insert$_\mathcal{P}$ $(\mathsf{Name}(p), \mathsf{Ref}(p))$**.** Suppose that site q is handling this request. First, q executes $\mathsf{Search}_\mathcal{P}$ $(\mathsf{Name}(p))$, which returns a reference to the closest predecessor q' of p in $\mathcal{P}$. q will then ask q' to integrate p in $\mathcal{P}$ (using the given topology control scheme). Once p has been integrated, p will ask q' to execute $\mathsf{Insert}_\mathcal{F}(\mathsf{Name}(p), \mathsf{Name}(p))$ (i.e. the reference to p is its random name) so that p has a handle in $\mathcal{F}$. Finally, objects in $\mathcal{F}$ stored at its neighbors will be moved to p so that we are back to a consistent mapping of $\mathcal{F}$ to $\mathcal{P}$. (Notice that this will give p at least one object, because $(\mathsf{Name}(p), \mathsf{Name}(p))$ is an object in $\mathcal{F}$, allowing it to get access to $\mathcal{F}$.)

Delete$_\mathcal{P}$ $(\mathsf{Name}(p))$**.** Suppose that site q is handling this request. Then q executes $\mathsf{Delete}_\mathcal{F}(\mathsf{Name}(p),$ $\mathsf{Name}(p))$ and afterwards removes p from $\mathcal{P}$ (using the given topology control scheme).

Search$_\mathcal{P}$ (Name)**.** Here the search strategy of the topology control scheme selected for $\mathcal{P}$ is used.

A topology for $\mathcal{P}$ may be established by using the Chord system or the Hyperring.

Operations for the file structure $\mathcal{F}$

Since $\mathcal{F}$ is only a concurrent data structure, it suffices to implement the three operations $\mathsf{Insert}_\mathcal{F}$ $(\mathsf{Name}(o), \mathsf{Ref}(o))$, $\mathsf{Delete}_\mathcal{F}(\mathsf{Name}(o))$, and $\mathsf{Search}_\mathcal{F}(\mathsf{Name})$. The concurrent Hyperring scheme may be used here.

Site operations

Finally, we explain on a high level how to execute the five site requests $\mathsf{Join}()$, $\mathsf{Leave}()$, $\mathsf{Insert}()$, $\mathsf{Delete}()$, and $\mathsf{Search}()$. Suppose that p is the site issuing the request.

- $\mathsf{Join}(\mathsf{Ref}(p))$: This will call $\mathsf{Join}_\mathcal{P}(\mathsf{Ref}(p))$.
- $\mathsf{Leave}(p)$: This will call $\mathsf{Leave}_\mathcal{P}(p)$.
- $\mathsf{Insert}(\mathsf{Name}(o), \mathsf{Ref}(o))$: First, p retrieves object o with the help of $\mathsf{Ref}(o)$ (or simply creates an object with its reference, depending on the application). Then it selects a random virtual name $\mathsf{Virtual\_Name}(o) \in [0,1)$ for o and calls $\mathsf{Search}_\mathcal{P}(\mathsf{Virtual\_Name}(o))$, receiving a reference to some site q. Then p will ask q to store o and to execute $\mathsf{Insert}_\mathcal{F}(\mathsf{Name}(o), \mathsf{Virtual\_Name}(o))$. q will do this by selecting a random object o' in $\mathcal{F}$ stored in it as starting point (taking the object representing q in $\mathcal{F}$ only if no other is available).
- $\mathsf{Delete}(\mathsf{Name}(o))$: p chooses a random object o' in $\mathcal{F}$ stored in p and executes $\mathsf{Delete}_\mathcal{F}(\mathsf{Name}(o))$ starting with o'.
- $\mathsf{Search}(\mathsf{Name}))$: p chooses a random object o' in $\mathcal{F}$ stored in p and executes $\mathsf{Search}_\mathcal{F}(\mathsf{Name})$ from there.

Using the concurrent Hyperring scheme with some extras, these operations can be implemented to obtain the following result:

Theorem 12.2 ([1]) *The above operations can be implemented with time and work complexity $O(\log^{O(1)} n)$, where n is an (unknown) bound on the cardinality of set* Sites *during the lifetime of the system.*

12.3 Data balancing

Next we consider the situation in which data is assigned to sites using their real names. Suppose that we use to concurrent Hyperring scheme to organize the sites. For simplicity, we assume that the data space is represented by the interval $[0, 1)$.

Every site in our topology control scheme is now represented by a *cluster* of sites of size $\Theta(\log n)$. Every site that wants to join the system is sent to a random cluster by using the first phase of Contact. This makes sure that new sites are distributed uniformly at random (up to a small constant factor) among the clusters, and therefore an oblivious adversary with bounded change rate cannot kill too many sites in a cluster in a short amount of time.

Each cluster owns a certain range of the data space, and this range represents its name in the DNS scheme presented in the previous section. I.e. if cluster C owns the range $[0.2, 0.5]$, $\mathsf{Name}(C) = [0.2, 0.5]$. As in the DNS scheme, our aim is to keep the ranges in sorted order around the 0-ring to ensure that queries can be efficiently routed at any time. To ensure a high robustness of our searchable data structure, mechanisms are needed so that

1. every cluster has a size of $\Theta(\log n)$ at any time,
2. the data load is evenly distributed among the nodes of a cluster (up to a constant factor), and
3. the data load is evenly distributed among the clusters (up to a constant factor).

In the following subsections we show how to solve each of these points. We start with the description of a co-called split-and-merge strategy for the first point.

Split and merge

Our aim is to ensure that every cluster has to contain at least $2w$ and at most $10w$ nodes at any time, where $w = \Theta(\log n)$ will be specified later. A cluster violating these bounds is called *bad*. A cluster is called *dangerous* if it contains less than $3w$ nodes or more than $9w$ nodes, and a cluster is called *safe* if it contains between $4w$ and $8w$ nodes. During the course of time, a cluster may lose or get nodes. In order to make sure no cluster ever gets bad, we use the following rules.

Recall the ring numbering strategy used for concurrent updates. The clusters are organized in groups with the help of the 1-edges belonging to the 1-ring with number 0. For every 1-edge e that is bridging one 0-edge, we take e and its succeeding 1-edge e' and combine all clusters bridged by e and e' (including the starting point of e and excluding the endpoint of e') into a group. For all other 1-edges e belonging to the 1-ring with number 0, we combine all clusters bridged by e (including the starting point of e and excluding the endpoint of e) into one group. These rules ensure that groups either consist of two or three clusters. Each of these groups is handled independently when performing split-and-merge operations. This is done in the following way:

Suppose that the group consists of only two clusters, C_1 and C_2. If none of the clusters is dangerous, we do nothing. Otherwise, if it is possible to move nodes between the two clusters so that none is dangerous afterwards, this is done with a minimum number of movements necessary for this. If this is not possible, then we are left with the following two cases:

- $|C_1| + |C_2| \leq 6w$: merge C_1 and C_2 into a single cluster, which creates a deletion event for the network. The resulting cluster is safe, since C_1 and C_2 were not bad.

- $|C_1|+|C_2| \geq 18w$: split the larger of the two clusters into two clusters, which creates an insertion event for the network, and move a minimum number of nodes from the remaining cluster to the new clusters so that the remaining cluster is non-dangerous. The two new clusters are safe, since C_1 and C_2 were not bad.

Now consider the case that the group consists of three clusters, C_1, C_2, and C_3. If none of the clusters is dangerous, we do nothing. Otherwise, if it is possible to move nodes between the two clusters so that none is dangerous afterwards, this is done with a minimum number of movements necessary for this. If this is not possible, then we are left with the following two cases:

- $|C_1| + |C_2| + |C_3| < 8w$: merge the clusters into a single cluster, creating two deletion events. The new cluster is safe, because none of the clusters was bad.
- $|C_1| + |C_2| + |C_3| \in [8w, 9w]$: merge the pair out of (C_1, C_2) and (C_2, C_3) with the smallest number of nodes into one cluster, creating one deletion event, and move a minimum number of nodes from the new cluster to the remaining cluster so that the remaining cluster is non-dangerous. Also here the new cluster is safe.
- $|C_1|+|C_2|+|C_3| \geq 27w$: split the largest cluster into two clusters, creating one insertion event, and move a minimum number of nodes from the old clusters to the new clusters so that the old clusters are non-dangerous. The new clusters are safe.

These rules immediately result in the following lemma.

Lemma 12.3 *The split-and-merge scheme ensures that as long as no cluster becomes bad, afterwards no cluster will be dangerous. Furthermore, in any event in which a cluster is split or merged, the resulting clusters are safe.*

Load balancing within a cluster

Next we explain how the load is stored and kept balanced in a cluster. Each cluster is organized like a cell: $2w$ of its nodes form its *core* and the rest its *periphery*. All the data is stored in its core, and the nodes in its periphery are free resources that can be moved between clusters.

In the core, the data is stored in sorted order among the nodes. So each of the nodes has its own data range. The load is kept balanced by using a load threshold ℓ. If insertions of new data cause a node u to have a load of more than 3ℓ, then a pair of neighboring nodes (w.r.t. their ranges) is sought with load $\leq 2\ell$. The lighter loaded one in this pair moves its data and range to the heavier loaded one and then accepts ℓ load (and the corresponding range) from node u. There is always such a pair as long as for the total data load L in the core we have $L \leq 2w \cdot \ell$. If this is not the case, then ℓ is increased to 2ℓ. If due to deletions of data, $L \leq (w/2) \cdot \ell$, then ℓ is reduced to $\ell/2$.

It is easy to check that any an increase in ℓ does not cause any data movements and between two decreases in ℓ, $\Omega(w \cdot \ell)$ data must have been deleted. Furthermore, the amount of data that must be inserted between two increases of a node to a load of more than 3ℓ must be at least ℓ. Hence, we can charge movements of data for the load balancing to insertions and deletions of data so that the following result holds:

Lemma 12.4 *For any sequence of insertions and deletions of data in a cluster, the cluster balancing scheme is $O(1)$-competitive concerning the data movements (including insertions of new data) a best possible balancing scheme (that does not require the data to be sorted) would have to perform to keep the data evenly balanced among the nodes of the cluster at any time.*

Load balancing in split and merge

Next we explain how data is moved when applying the split-and-merge rules. There are three different cases to consider:

- A node is moved from cluster C_1 to cluster C_2: If C_2 is the successor of C_1, then the largest ℓ data items in C_1 (and the range corresponding to them) are moved from C_1 to C_2, where ℓ is the average load over all nodes of C_1 (including the periphery). If C_2 is the predecessor of C_1, then the smallest ℓ data items in C_1 are moved from C_1 and C_2. Otherwise, it may happen in a group of three clusters that C_1 and C_2 are not direct neighbors. Suppose w.l.o.g. that C_2 is a successor of C_1. In this case, we move a node directly from C_1 to C_2 but "tunnel" the data from C_1 to C_2 along the clusters in between, say $C'_1, \ldots, C'_k$. This is done in a way that first C_1 sends its ℓ largest data items to C'_1, then C'_1 sends its ℓ largest data items to C'_2, and so on, until C'_k sends its ℓ largest data items to C_2. This preserves the sorted order of the data. We note that k can be at most 3.

- A cluster C is split into two clusters C_1 and C_2: In this case, C splits its range at the median into two ranges and splits its core accordingly so that one part is given to C_1 and the other to C_2. Also, the periphery is split so that C_1 and C_2 have the same number of nodes (up to possibly one). We note that up to one node in the core that has data of both ranges, no data movements are caused by a split. (However, the adjustment of the load threshold in the new clusters may cause data movements.)

- Two clusters C_1 and C_2 are merged into one cluster C: In this case we take any one of the two that is dangerous, say C_1 (w.l.o.g.), and move all of its data to the core of C_2. Afterwards, all nodes in C_1 join the periphery of C_2. Thus, merging two clusters involves only data movements from a dangerous one.

Our split-and-merge scheme has the crucial property that

- only a dangerous cluster is split, and the outcome are two safe clusters, and

- only a dangerous cluster is merged into another one, and the outcome is a safe cluster.

Since there are at least w nodes difference between a dangerous cluster and a safe cluster, any cluster involved in data movements in two split or merge operations must have gained or lost at least w nodes in between. Hence, we get the following result.

Lemma 12.5 *For any sequence of insertions and deletions of nodes in the clusters that do not cause them to become bad, the split-and-merge balancing scheme is $O(1)$-competitive concerning the data movements an optimal balancing scheme (that does not require the data to be sorted) would have to perform to keep the data evenly balanced among all nodes in each cluster at any time.*

Combining this with Lemma 12.4, we obtain:

Lemma 12.6 *For any sequence of insertions and deletions of data or nodes in the clusters that do not cause them to become bad, the combined cluster and split-and-merge balancing scheme is $O(1)$-competitive concerning the data movements an optimal balancing scheme (that does not require the data to be sorted) would have to perform to keep the data evenly balanced among all nodes in each cluster at any time.*

Load balancing across network edges

Certainly, the cluster balancing scheme and split-and-merge balancing scheme do not suffice to keep the load balanced across all clusters in the system. So we also perform balancing across the network edges. However, *the only thing we ever do here is moving nodes of the periphery between clusters*. I.e., no data item is ever moved when balancing across network edges. The movement of periphery nodes is done separately for each level of the Hyperring. The load balancing for level i works as follows:

Every cluster splits into two sub-clusters of equal number of nodes, one for each endpoint of the two i-edges in which it participates. For every i-edge e with sub-clusters C_1 and C_2 let m_j be the number of nodes in C_j and ℓ_j be the average load of C_j, $j \in \{1,2\}$. Suppose w.l.o.g. that $\ell_1 < \ell_2$. Then the number of periphery nodes moved from C_1 to C_2 is equal to

$$\min\left\{\left\lfloor\frac{(\ell_2-\ell_1)\min\{m_1,m_2\}}{\ell_1+\ell_2}\right\rfloor,\ w/2\right\}\ .$$

The first term in this expression makes sure that for the average loads ℓ'_j after the balancing it holds $\ell'_1 \leq \ell'_2$, and the second term makes sure that no sub-cluster (and therefore no cluster) becomes bad as long as no node leaves or enters the system. If nodes enter or leave the system, then we assume that w is chosen so that this is limited to $w/2$ between any two applications of the split-and-merge balancing scheme. In this case, we would have to replace $w/2$ in the formula by $w/4$.

Since no data movements are performed, no competitive ratio needs to be provided here.

The complete load balancing scheme

The load balancing is performed in rounds. Each round consists of d phases numbered from 0 to $d-1$, where d is the maximum degree of a cluster in the Hyperring. Phase i consists of two steps:

1. Execute the split-and-merge balancing scheme to make all clusters non-dangerous.

2. Execute the cluster balancing scheme to keep the load within a cluster evenly balanced up to a constant factor.

3. Execute the network balancing scheme for all i-edges.

Note that the clusters do not have to know d. Each cluster goes through its own phases based on its degree. However, if it wants to perform load balancing across some i-edge e, then it has to wait for the other cluster connected to e to be also in phase i before load balancing can be performed. This keeps the phases (quasi-)synchronized among the clusters and therefore allows to execute a round as described above.

Next we state some theorems about the performance of this balancing scheme. Because their proofs are quite involved, we omit them. First we show that when the load balancing scheme terminates, then there is only a constant factor deviation in the load of every node.

Theorem 12.7 *When the load balancing scheme terminates, the load of the nodes only deviates by a factor of at most* $\mathrm{e}^{(3\log n)/w}$.

Hence, if $w = \Omega(\log n)$ we only have a constant factor deviation in the load. Next we show how much work and time it needs to achieve this load balance. Let the initially maximum load of a node be defined as $\ell_{\max}$ and the average load of a node be defined as $\bar{\ell}$.

Theorem 12.8 *The load balancing scheme is* $O(w\log(\ell_{\max}/\bar{\ell})$*-competitive concerning the data movements it takes to get to a load distribution in which the maximum and minimum load of a cluster only deviates by a constant factor.*

Theorem 12.8 is very strong, because for any constant degree network there are load distributions so that *any* balancing strategy that can only move data along the edges needs $\Omega(\log n \cdot \log(\ell_{\max}/\bar{\ell}))$ data movements to distribute the load evenly among all nodes.

Finally, we state a bound on the time it takes to arrive at a load distribution where the maximum load is at most a constant times the average load. We assume here that $w = \Omega(\log n)$.

Theorem 12.9 *The load balancing needs at most* $O(\log n)$ *rounds to arrive at a load at a load distribution where the maximum load is at most a constant times the average load.*

References

[1] B. Awerbuch and C. Scheideler. Self-repairing and searchable distributed data structures. Dept. of Computer Science, Johns Hopkins University, November 2002.

[2] A. Fiat and J. Saia. Censorship resistant peer-to-peer content addressable networks. In *Proc. of the 13th ACM/SIAM Symp. on Discrete Algorithms (SODA)*, 2002.

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

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, 2001.

[5] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[6] B. 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.