Novel Architectures for P2P Applications: the Continuous-Discrete Approach*†

MONI NAOR‡
The Weizmann Institute of Science
moni.naor@weizmann.ac.il

UDI WIEDER§
Microsoft Research
uwieder@microsoft.com

October 4, 2006

Abstract

We propose a new approach for constructing P2P networks based on a dynamic decomposition of a continuous space into cells corresponding to servers. We demonstrate the power of this approach by suggesting two new P2P architectures and various algorithms for them. The first serves as a DHT (Distributed Hash Table) and the other is a dynamic expander network. The DHT network, which we call Distance Halving, allows logarithmic routing and load, while preserving constant degrees. It offers an optimal tradeoff between the degree and the path length in the sense that degree d guarantees a path length of $O(\log_d n)$. Another advantage over previous constructions is its relative simplicity. A major new contribution of this construction is a dynamic caching technique that maintains low load and storage even under the occurrence of hot spots. Our second construction builds a network that is guaranteed to be an expander. The resulting topologies are simple to maintain and implement. Their simplicity makes it easy to modify and add protocols. A small variation yields a DHT which is robust against random Byzantine faults. Finally we show that, using our approach, it is possible to construct *any* family of constant degree graphs in a dynamic environment, though with worse parameters. Therefore we expect that more distributed data structures could be designed and implemented in a dynamic environment.

1 Introduction

The problem we deal with is how to maintain a network structure and functionality in a dynamic environment where the participants of the network change over time. Such networks are sometimes called Peer-to-peer (P2P) systems. Peer-to-peer networks are characterized by a mostly symmetric communication pattern and by the lack of central control or a-priori hierarchical organization. Moreover a P2P system is expected to scale gracefully as the size of the network grows. The sheer scale and dynamism in which P2P networks are supposed to operate make the design of P2P systems challenging even for relatively simple applications.

A *distributed hash table* (DHT) is a giant hash table that is maintained by a large number of servers in a P2P manner. The hash table interface is useful for the implementation of a large variety of tasks, therefore it received a considerable amount of attention from the research community. Previous DHT designs include the Plaxton-Rajaraman design [39], Tapestry [48], Chord [45], Pastry [43], CAN [41], Kademlia [32], Viceroy [29] and many more. These systems follow the general paradigm of *consistent hashing* [19]: Let I

*Preliminary versions of the paper appeared in [36] and [35].

†Research supported in part by the RAND/APX grand from the EU Program IST

‡Incumbent of the Judith Kleeman Professorial Chair.

§Research done while at the Weizmann Institute denote the space into which the data item's keys are hashed. The idea is to assign to each node an ID from I as well. Typically $I = \{0,1\}^k$ for some $k > 0$, without loss of generality we may assume that $I = [0, 1)$. Assign each node a segment (say the segment that lies between its ID and the next larger ID) and let each node be responsible for storing all the data items with hash values that fall within its assigned segment. The connections in the network are also determined by the ID's of the different nodes. Connections are set such that the system supports a *lookup* protocol that allows nodes to find the node which is responsible for a required hash value, and thus retrieve a data item. The differences between the various DHT's lie in the different ways in which the connections are established, and the different algorithms by which the routing paths are found[1].

The methodology used in designing these networks could be roughly described as follows: first find a *static family* of graphs in which there are good protocols for performing the desired tasks. Typically designers aim at one of the classical inter-connection networks such as hypercubes and grids. The next task is to show how to construct in a distributed manner a network with a topology that 'approximates' the topology of the static family of graphs. In this sense CAN approximates the $d-$dimensional torus. Chord and Pastry approximate the hypercube and Viceroy approximates the butterfly. The continuous-discrete approach gives a unified technique for performing this. We use the continuous-discrete approach to design DHTs based on the De-Bruijn graph which we call *Distance Halving*.

The parameters by which a DHT is measured include the following metrics:

- **Cost of join/leave:** The service should accommodate changes easily. When servers join or leave, only a small number of servers should change their state. In particular the degree of the graph that represents the network, should be small.
- **Congestion:** No server should be a bottleneck on the performance of the service. The load incurred by lookups routing through the system should be evenly distributed among participating servers. See Section 2.2 for a formal definition.
- **Lookup path length:** The forwarding path of a lookup should involve as few machines as possible. We aim to minimize the maximum path length in the network.
- **Fault tolerance:** The service should function well after some of its servers or connections fail. We should consider the scenario in which a random subset of the servers fail, and the worst case scenario in which an adversary chooses which servers fail. In each of these cases there are two models to consider. The first is the fail and stop model in which failed servers/connections do not respond at all. The second is a Byzantine model in which failed servers may act in an inconsistent and malicious way.
- **Dynamic caching:** Highly popular data items may cause a bottleneck at and around their location. Relieving the congestion around the hot spot requires the service to support some dynamic caching mechanism, in which the data item is replicated to other servers. We want to allow the maximally congested server in the system to have a low load while maintaining the number of data items each server has to store as small as possible.

Table 1 summarizes the performance of different constructions under these parameters.

[1] There may be various ways in which a lookup service is implemented even when the network is given and fixed. For instance in 'real life' systems, an iterative lookup algorithm may behave very differently from a recursive one. We are interested in the algorithmic/combinatorial nature of the algorithms and ignore such issues.

Lookup Scheme	*path length*	*congestion*	*linkage*
Chord [45]	$\log n$	$(\log n)/n$	$\log n$
Tapestry [48]	$\log n$	$(\log n)/n$	$\log n$
CAN [41]	$dn^{1/d}$	$dn^{1/d-1}$	d
Small Worlds [22]	$\log^2 n$	$(\log^2 n)/n$	$O(1)$
Viceroy [29]	$\log n$	$(\log n)/n$	$O(1)$
Distance Halving (ours)	$\log_d n$ $(2 \leq d \leq \sqrt{n})$	$(\log_d n)/n$	$O(d)$

Table 1: Comparison of *expected* performance measures of lookup schemes.

1.1 Our Contributions

Our Contributions are both in the conceptual and in the concrete levels. Conceptually we provide a set of design rules and a framework in which we believe it is relatively simple to design and analyze dynamic data structures. We call the approach 'continuous-discrete'. It is the first attempt to unify different constructions in this field.

Concretely we suggest five novel constructions and algorithms:

- We present a novel construction for a DHT in Section 2. The construction is very simple and offers logarithmic dilation and load. An important feature is that it has an optimal tradeoff between the degree and the dilation. A degree of d guarantees a dilation of $O(\log_d n)$. Previous constructions either had a logarithmic degree (such as Chord [45]) or were more involved (such as Viceroy [29]). Thus if one is interested in constant degree DHTs, then the simplicity of our system wins over Viceroy and if one is interested in $\log n$ degree overlay networks, then our construction has shorter path lengths. See Table 1.

  Our DHT construction is inspired by the De-Bruijn graph. We are not the only ones to use the De-Bruijn graph in this context. Constructions using it were suggested independently by Fraigniaud and Gauron [12], Kaashoek and Karger [18] and Abraham *et al.*[1]. The parameters they achieve are similar to those proved in Section 2, yet their approach in emulating the De-Bruijn graph is different. They try to do it directly whereas we use the continuous-discrete approach to emulate an 'infinite version' of the De-Bruijn graph. In particular it is not clear how to obtain in these constructions a caching protocol which relieves hot spots and a low load in permutation routing, which we show in the following sections.

- We show a dynamic caching algorithm in Section 3 that provably ensures that under *any* set of requests for data items which are independent of the intrinsic choices of the hash functions, all servers enjoy low load with high probability. Thus it relieves the occurrence of hotspots. Dynamic caching achieved a considerable amount of attention under many different models. The problem of dynamic caching in DHT's was specifically raised by Ratnasamy *et al.*at [42]. To the best of our knowledge the algorithm we present is the first to ensure this property.

- In Section 4 we present several algorithms for maintaining a good load balancing between servers. These techniques allow us to build DHT's with constant degrees *with high probability*. One of the methods we show guarantees constant degrees in the worst case. Other constructions with constant average degree [12][18] have maximum degree of $O(\log n)$. The techniques we show are applicable for other DHT constructions as well.

- In Section 5 we show a distributed construction of a network which is guaranteed to be a constant degree expander. To the best of our knowledge our construction is the first to produce constant degree expanders in a distributed way[2]. Possible applications for dynamic expanders include load balancing jobs and an infrastructure for maintaining probabilistic quorums.
- The DHT construction is very flexible. In Section 6 we show how a slight variation of it is robust against random faults of the servers.
- Finally a general technique for emulating *any* graph in a distributed setting is shown in Section 7. This may be used for designing more dynamic data structures.

We stress that the simplicity of the continuous-discrete approach plays a central role in the design and analysis of the algorithms above. In particular in that of the caching algorithm of Section 3 and the fault tolerant construction of Section 6. We see the relative ease in which these problems were solved as a 'proof of concept' for the entire approach.

1.2 The Continuous-Discrete Approach

We present a high-level description of the framework which may be titled "think continuously, act discretely". Let I be a Euclidean space. Let G_c be a graph where the vertex set is some continuous set I. Each point in I is connected to some other points. The actual network is a *discretization* of this continuous graph based on a dynamic decomposition of the underlying space I into cells where each server is responsible for a cell. Two cells are connected if they contain adjacent points in the continuous graph. The partition of the space into cells should be maintained in a distributed manner. When a Join operation is performed an existing cell splits, when a Leave operation is performed two cells are merged into one. In our view the task of designing a dynamic and scalable network should follow the following design rules:

1. Choose a proper continuous graph G_c over the continuous space I. Design (and prove the correctness of) the algorithms in the *continuous* setting. Designing the algorithms in the continuous graph is typically quite simple. It has the advantage of ignoring the scalability issue, and it offers strong and simple mathematical tools for proving statements.
2. Find an efficient way to discretize the continuous graph in a distributed manner, such that the algorithms designed for the continuous graph would perform well in the discrete graph. The discretization is done via a decomposition of I into cells. An important parameter of the decomposition of I is the ratio between the size of the largest and the smallest cell which we call the *smoothness*. We show that a decomposition in which the smoothness is constant can be used to build the Distance Halving DHT and the expander based networks. In Section 6 we show that if the cells which compose I are allowed to *overlap* then the resulting graph would be fault tolerant.

The main advantage of the approach is that it gives a *unified* method for performing the Join/Leave operation and dealing with the scalability issue, thus separating it from the actual network and allowing a more *modular* design. In the following chapters we demonstrate the usefulness of the approach.

[2]Law and Siu have independently designed another algorithm which builds an expander with high probability [25]. Their approach is completely different from ours.

2 The Distance Halving DHT

2.1 The Construction

First we define the *continuous* Distance Halving graph G_c. The vertex set of G_c is the interval $I \stackrel{def}{=} [0,1)$. The edge set of G_c is defined by the following functions: $\ell(y) \stackrel{def}{=} \frac{y}{2}$, $r(y) \stackrel{def}{=} \frac{y}{2} + \frac{1}{2}$ where $y \in I$, ℓ abbreviates 'left' and r abbreviates 'right'. Note that if y is written in binary, then $\ell(y)$ effectively inserts 0 into the left and $r(y)$ shifts a 1 into the left. This property is the reason the graphs is similar to classic inter-connection networks and lies at the heart of the routing algorithms.

The out-degree of each point is 2 while the in-degree is 1. It would often be the case that we treat the graph as undirected. In this case we call the single incoming edge the *backward edge* and denote it as $b(y) \stackrel{def}{=} 2y \mod 1$. In Figure 1 the upper diagram shows the edges of a point in I, the lower diagram shows that an interval is mapped into two intervals, each half its size. We may abuse the notation and write $r([y,z)), \ell([y,z))$ meaning the image of the interval $[y,z)$ under r, ℓ. Next we show how to construct a *discrete* Distance Halving graph. The nodes of the graph correspond to a set of n servers $V_0, V_1, \ldots, V_{n-1}$. Denote by $\vec{x}$ a set of n points in I such that $x_0 < x_1 < \ldots < x_{n-1}$. The point x_i would typically be a hash of the i.d of some server V_i. The points of $\vec{x}$ divide I into n segments. Define the *segment* of x_i to be $s(x_i) = [x_i, x_{i+1})$ $(i = 1 \ldots n-1)$ and $s(x_n) = [x_{n-1}, 1) \cup [0, x_1)$.

In the *discrete* Distance Halving graph $G_{\vec{x}}$, each server V_i is associated with the segment $s(x_i)$, we may refer to this segment as $s(V_i)$ as well. If a point y is in $s(x_i)$ we say that V_i *covers* y. A pair of vertices (V_i, V_j) is an edge of $G_{\vec{x}}$ if there exists an edge (y,z) in the *continuous* graph, such that $y \in s(x_i)$ and $z \in s(x_j)$. The edges (V_i, V_{i+1}) and (V_{n-1}, V_0) are added such that $G_{\vec{x}}$ contains a ring. The ring edges are anti-parallel directed edges. If a new server V wishes to enter the system it does the following:

Algorithm Join

1. Choose some point x and set $V.id \leftarrow x$. (How to choose x is not shown here. Various options for doing that are discussed in Section 4).
2. Call the lookup procedure[3] for point x. The procedure returns the address of the server V_j for which $x \in s(x_j)$.
3. The segment $s(x_j)$ should be divided into two parts so that $s(x) = [x, x_{j+1})$. Receive from V_j all the data items that are mapped to $s(x)$ and the addresses of all the neighbors V should have.
4. Inform the neighbors of V so that they can change their address tables accordingly.

Assuming the graph has constant degree, Step 3 of the algorithm involves only $O(1)$ messages. Step 2 of the algorithm involves one operation of Lookup which would be discussed later. Step (1) has some variations which are discussed in Section 4. When server V_i leaves the network, some existing server should take over its segment. The simplest solution would be that the server that is the predecessor of V_i on the ring enlarges its segment such that it includes $s(x_i)$. More sophisticated solutions are discussed in Section 4.

Theorem 2.1. *The total number of edges in $G_{\vec{x}}$ without the ring edges is at most $3n-1$.*

[3]We do not deal with the problem of how to perform this initial lookup. It is assumed that that the server has some some host server which helps it to join the system.

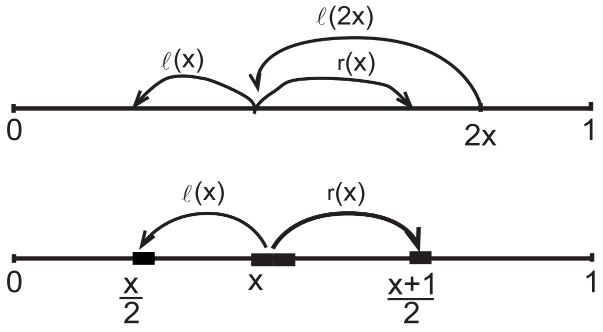

Figure 1: The upper figure demonstrates the edges of a point in the continuous graph. The lower figure shows a mapping of a segment to two smaller ones.

Proof. The proof is by induction on n. If $n = 1$ then there are two self edges.

Now assume the segment $[x, z)$ is split into $[x, y), [y, z)$. The segment which covers the point $y/2$ will be the only segment into which both segments $[x, y)$ and $[y, z)$ will have a left edge, thus at most one new *left* edge is formed. Similarly there would be at most one new right edge and one new backward edge. $\square$

Theorem 2.1 implies that for every vector $\vec{x}$, the *average* degree of the graph is at most 6. In order to bound the *maximum* degree of the graph, another property should be considered:

Definition 1. *The smoothness of $\vec{x}$ is denoted by $\rho(\vec{x})$ and is defined to be* $\max_{i,j} \frac{|s(x_i)|}{|s(x_j)|}$.

If it is guaranteed that the smoothness of $\vec{x}$ is bounded by some constant independent of n, we say that $\vec{x}$ is smooth. We may abuse the notation and write $\rho(G_{\vec{x}})$ instead of $\rho(\vec{x})$. The smoothness of $\vec{x}$ plays a central role in the analysis of the construction.

Theorem 2.2. *The maximum out-degree of $G_{\vec{x}}$ without the ring edges is at most $\rho(\vec{x}) + 4$, the maximum in-degree without the ring edges is at most $\lceil 2\rho(\vec{x})\rceil + 1$.*

Proof. Let i be such that $|s(x_i)|$ is maximum. The length of the minimum segment is therefore at least $\frac{|s(x_i)|}{\rho}$. We have $|r(s(x_i))| = \frac{1}{2}|s(x_i)|$, therefore there are at most

$$\lceil (\tfrac{1}{2}|s(x_i)|)/(\tfrac{|s(x_i)|}{\rho})\rceil + 1 = \lceil \tfrac{1}{2}\rho\rceil + 1$$

different segments that intersect the interval $r(s(x_i))$. The same argument applies for $\ell(s(x_i))$, and we have $2(\lceil \frac{1}{2}\rho\rceil + 1) \leq \rho + 4$, which bounds the out-degree.

The bound on the in-degree follows in a similar way. Now the preimage of $|s(x)|$ is one contiguous of segment of length $2|s(x)|$, and at most $\lceil 2\rho(\vec{x})\rceil + 1$ different segments might intersect it. $\square$

Mapping the data items to servers The mapping of data items to nodes is done in the same manner as other constructions of distributed hash tables (such as consistent hashing [19], Chord [45], Viceroy [29] and CAN [41]). First data items are mapped into the interval I using a hash function. Server V_i should hold all data items mapped to points in $s(V_i)$. We assume that h is some hash function (for instance a $k-$wise independent function for some k), which is chosen at the construction of the system and is given to every server upon joining.

The De-Bruijn Graph The Distance Halving construction resembles the well known De-Bruijn graph.

Definition 2. *The r-dimensional De-Bruijn graph consists of 2^r servers and 2^{r+1} directed edges. Each node corresponding to an $r-$bit binary string. There is a directed edge from each node $u_1 u_2 \cdots u_r$ to nodes $u_2 \cdots u_r 0$ and $u_2 \cdots u_r 1$.*

The Distance Halving DHT emulates the De-Bruijn graph in the following sense. Assume that $n = 2^r$. Let $\vec{x}$ be a set of m points such that $x_i = \frac{i}{n}$, the discrete Distance Halving graph $G_{\vec{x}}$ without the ring edges is isomorphic to the r-dimensional De-Bruijn graph. To see this expand the interval I by a factor n so that $I = [0, 2^r)$. Now the location of point x_i is at i. Let $v_1 v_2 \ldots v_r$ be the binary representation of i. It is easy to verify that $\ell(x_i)$ is $0 v_1 \ldots v_{r-1}$ and that $r(x_i)$ is $1 v_1 \ldots v_{r-1}$. Now the isomorphism follows by mapping each $v_1 v_2 \ldots v_r$ of $G_{\vec{x}}$ to $v_r v_{r-1} \ldots v_1$ in the De-Bruijn graph.

The r-dimensional De-Bruijn graph is a well investigated combinatorial object. It is known for instance that its diameter and mixing time are $\Theta(r)$ and that the smallest bisection contains $\Theta(r^{-1} 2^r)$ edges. The ease in which short routes are found makes it a popular topology for parallel algorithms. See Leighton [26] for an overview of various properties of this graph.

While the definition of De-Bruijn graph we presented assumes each node is represented by a binary string, it is natural to generalize the definition so that each node is represented by a string of alphabet size Δ. In this case the diameter of the graph is $\log_\Delta n$. A variation of the continuous graph emulates the De-Bruijn graph with alphabet size Δ. See more details in Section 2.3. Different ways to emulate the De-Bruijn graph in a P2P manner were suggested in [12, 18, 1].

2.2 The Lookup Operation

We set some notation that is useful in the future. For any two points $x, y \in I$, define $d(x, y)$ to be $|x - y|$. Let σ denote some infinite sequence of binary digits, and σ_t denote its prefix of length t. Denote by $\sigma_t.0$ and $\sigma_t.1$ the concatenation of a bit to the string σ_t. For every point $y \in I$ we define the function $w(\sigma_t, y)$ in the following recursive manner:

$$w(\sigma_0, y) = y \tag{1}$$
$$w(\sigma_t.0\,,\ y) = \ell(w(\sigma_t, y)) \tag{2}$$
$$w(\sigma_t.1\,,\ y) = r(w(\sigma_t, y)) \tag{3}$$

In other words $w(\sigma_t, y)$ is the point reached by a walk that starts at y and proceeds according to σ_t from the least significant bit to the most significant bit of σ_t when 0 represents ℓ and 1 represents r.
Routing properties of the continuous DH graph: The following observation justifies the name 'Distance Halving':

Observation 2.3 (distance halving property)**.** *For all $y, z \in I$ and for all binary strings σ it holds that:*

$$d(r(y), r(z)) = |\tfrac{y}{2} + \tfrac{1}{2} - \tfrac{z}{2} + \tfrac{1}{2}| = \tfrac{1}{2} d(y, z)$$
$$d(\ell(y), \ell(z)) = |\tfrac{y}{2} - \tfrac{z}{2}| = \tfrac{1}{2} d(y, z)$$

therefore by induction on t

$$d(w(\sigma_t, y), w(\sigma_t, z)) \;=\; 2^{-t} \cdot d(y, z)$$

Observation 2.3 says that any binary string could be used to identify paths starting at y and at z that approach one another. This fact would be used to analyze the distance halving routing algorithm described in Section 2.2.2.

In the following we show a different way which yields short paths between different segments of the continuous graph.

Let $\sigma(y)$ be the binary representation of y and $\sigma(y)_t$ the first t bits of $\sigma(y)$.

Claim 2.4. *Let $y, z \in I$ .For all t it holds that $d\left(y, w(\sigma(y)_t, z)\right) \leq 2^{-t}$.*

The claim states that a walk determined by the binary representation of y would approaches y quickly, and this is *independent of the starting point z*.

Proof. Let h_t be the point reached by walking *backwards* from y (i.e. along the backward edges as defined in Section 2.1) for t steps. Note that the direction (left or right) of the i'th step in this walk is determined by the i'th bit in $\sigma(y)$. In other words it holds that: $w(\sigma(y)_t, h_t) = y$. We have:

$$d\left(w(\sigma(y)_t, z), y\right) = d\left(w(\sigma(y)_t, z), w(\sigma(y)_t, h_t)\right).$$

By Observation 2.3 it holds that $d\left(w(\sigma(y)_t, z)\,,\, w(\sigma(y)_t, h_t)\right) = 2^{-t} d(z, h_t) \leq 2^{-t}$. □

The previous two claims demonstrate two ways in which nodes of the continuous graph can approach one another. Loyal to our design rules we use the properties of the continuous graph to design simple routing algorithms on the *discrete* graph. These algorithms emulate the implied by Claims 2.3 and 2.4. Assume server V_i wishes to lookup the point y. The lookup should return the server V_j such that $y \in s(x_j)$. We present two algorithms that perform lookup. The first will have short lookup paths, while the second will increase the lookup path by a factor of at most two and will have better load balancing qualities.

2.2.1 Fast Lookup

Claim 2.4 states that from every pair of points y, z, a path starting at y which is determined by the binary representation of z would approach z quickly. Assume server V_i wants to find y. Let the point z be the midpoint in $s(V_i)$. Now we know that for every number t, a path starting at y which walks according to $\sigma(z)_t$ would reach a point which is at distance 2^{-t} from z. So if t is large enough $w(\sigma(z)_t, y) \in s(V_i)$. A subtle point is that this path is determined by $\sigma(z)_t$ from the least significant bit towards the most significant, as a result the number t should be chosen in advance such that indeed $w(\sigma(z)_t, y) \in s(V_i)$. Let t be the smallest number such that $w(\sigma(z)_t, y) \in s(V_i)$ and denote $h := w(\sigma(z)_t, y)$. Now, a message travelling from V_i to y using Fast Lookup would start at h and travel the path backwards, along the backward edges, until it has reached y. The following is a more formal description of the protocol:

Fast Lookup (V_i, y)

1. Let z be the point in the middle of $s(V_i)$. Calculate the minimum t such that $w(\sigma(z)_t, y)$ is in $s(V_i)$.
2. Let $h = w(\sigma(z)_t, y)$. Start moving from h (i.e. from $s(V_i)$ in the continuous graph and therefore from V_i) on the *backward* edges. Each hop the header of the message should be updated so that it holds the current position of the message on I. After t steps the server covering y is reached.

The length of the lookup path is determined by t. By Claim 2.4 we need $2^{-t} \leq |S(V_i)|/2$ so $t = -\log(|s(V_i)|) + 1$ suffices. The shortest segment in I is of length at least $\frac{1}{\rho n}$, therefore we have:

Corollary 2.5. *The length of a lookup path taken by Greedy Lookup is at most* $\log n + \log \rho + 1$.

Note that in Fast Lookup the servers need not to know what ρ or n are, and all computations are based on local knowledge only. Next we analyze the congestion of Greedy Lookup.

Definition 3. *The congestion of server V_i is the probability V_i is active in a routing between a randomly chosen server and a random point. The congestion of the network is the maximum congestion over all its servers.*

First we prove that the congestion of the *continuous* graph is low.

Lemma 2.6. *Let y, z be chosen randomly from I. The probability that server V participates in a Fast Lookup of length t that starts at y and approaches z is at most $|s(V)|(t+1)$.*

Proof. We show that for every $j \geq 0$ the probability a message hits V after exactly j steps is at most $|s(V)|$. The theorem is then derived by union bounding the probabilities for $0 \leq j \leq t$. The message hits V at step 0 iff $y \in s(V)$ which happens with probability $|s(V)|$. Denote by $b(V)$ all points which are reached via the backward edges from $s(V)$. Clearly $|b(V)| = 2|s(V)|$. Now, in order for the message to hit V at the jth step it must hit $b(V)$ at step $j-1$. The probability of this $2|s(V)|$ by the induction hypothesis. Given that the message is at $b(V)$, the probability it hits $s(V)$ the next step is exactly $\frac{1}{2}$, due to the randomness of z. Thus the probability the message hits $s(V)$ at the jth step is at most $\frac{1}{2}2|s(V)| = |s(V)|$. □

Theorem 2.7. *Let V be a server, the congestion of V is at most $(\log n + \log \rho + 1)\rho|S(V)|$ which implies that if $\vec{x}$ is a smooth set of points, the congestion of $G_{\vec{x}}$ under Greedy Lookup is $\Theta(\frac{\log n}{n})$.*

Proof. The proof follows the same lines of the previous lemma. The number of hops in the path is bounded by $\log n + \log \rho + 1$. For each hop in the path, we show that the probability that V participates is bounded by $\rho|s(V)|$. Fix some $i \leq \log n + \log \rho + 1$. As before, there is a unique segment of length $2^i|s(V)|$ in which y must exist in order for the path to reach $s(V)$ on i^{th} step. The probability of this occurring is $2^i|s(V)|$. Now, given that y is in this segment, in order for V to participate, there is a unique set of i steps leading from y to $|s(V)|$. Let U be a randomly chosen server which lookups y and let z be its middle point. We need to calculate the probability the first i bits of $\sigma(z)$ are indeed i bits leading from y to $s(V)$. This is tantamount to asking what is the probability z falls within some segment of length 2^{-i}. Since the smoothness is ρ, the size of each $s(U)$ is at least $\frac{1}{\rho n}$, thus there are at most $2^{-i}\rho n$ different servers whose middle points might be in the segment. Thus the probability V participates in the i^{th} step is bounded by:

$$\frac{2^{-i}\rho n}{n} \cdot 2^i|s(V)| = \rho|s(V)|.$$

The second part of theorem follows by the fact that if ρ is constant then $|s(V)|$ is $\Theta(\frac{1}{n})$.

□

2.2.2 Distance Halving Lookup

The Distance Halving lookup scheme enjoys small congestion even in a worst case permutation routing. It has two phases, the first phase is to send the message to an almost random destination, the second phase routes the message from the random destination to the target. First we describe how to perform the first phase. When server V_i initiates a lookup for point y, it first chooses a random string of bits τ. The header of the message V_i sends should contain x_i which is V_is location, the target y, the random string τ, and a counter t initially set to 0. Upon receiving a message a server does the following:

Distance Halving Lookup - Phase I

1. Check if $w(\tau_t, y)$ is covered by the current segment or by one of the neighboring segments. If so move the message to the server which covers $w(\tau_t, y)$ and move to phase II.

2. Set $t \leftarrow (t+1)$ and update the header of the message.

3. Send the message to the neighbor covering the point $w(\tau_t, x_i)$. An edge exists in the continuous graph and therefore must exist in discrete graph as well.

In the second phase the message moves *backwards* along the backward edges from $w(\tau_t, y)$ to y. Each step the server handling the message deletes the last bit in τ.

It is convenient to think of the routing as if two messages are moving simultaneously, one from the source and one from the target. Both of them move according to the same sequence τ. Claim 2.3 states that each step the distance between them is shrinks in half, until their distance is at most $\frac{1}{\rho n}$, in which case they would be in the segment of the same server (or neighboring servers). The following theorem is therefore an immediate consequence of Claim 2.3:

Theorem 2.8. *The length of a lookup path taken by Distance Halving Lookup is at most* $2\log n + 2\log \rho$.

The following theorem states that the maximum congestion of the Distance Halving Lookup is also logarithmic.

Theorem 2.9. *Let $\vec{x}$ be a smooth set of points. The congestion of $G_{\vec{x}}$ under Distance Halving Lookup is* $\Theta(\frac{\log n}{n})$.

Proof. The first phase of the routing is similar to Fast Lookup, i.e. the messages pass through a random path. Thus the same analysis of Theorem 2.7 holds. The message then is in a random location and performs one step only which does not affect the congestion of the system. The second phase of the routing is analogical to the first phase. In the second phase the message traverses backward along a random path to the destination, thus the second phase is again identical to the Fast Lookup case and 2.7 holds for the second phase as well. □

2.2.3 Permutation Routing

The Distance Halving Lookup is similar in spirit to the routing scheme suggested by Valiant [46] for the hypercube, therefore it is not surprising that it imposes small congestion for worst case permutation routing. Let η be some permutation and assume that for all i server V_i initiates a lookup for a point in $s(V_{\eta(i)})$.

Theorem 2.10. *Given that $G_{\vec{x}}$ is smooth, then for every permutation η it holds that when routing η w.h.p each server participates in the routing of at most $O(\log n)$ messages, where the probability is taken over the random choices of the routing algorithm.*

Proof. First, note that since the length of each of the n lookup paths is $\Theta(\log n)$, by an averaging argument there is a server which serves $\Omega(\log n)$ messages.

For the upper bound fix a server V. In the following we prove that the *expected* number of lookups that V participates in is $O(\log n)$. The high probability bound will follow later. For a contiguous segment $Q \subset I$, let L_i^Q be the random variable counting the number of lookups that reach segment Q at the i^{th} step[4], and consider only the first phase of the routing. We claim that

$$\mathbb{E}\left[L_i^{s(V)}\right] \leq n(\rho+1)|s(V)|.$$

We prove it by induction on i. As common in proofs by induction we need to strengthen the claim slightly and prove that *any* segment $Q \subset I$, has $\Theta(n(\rho+1)|Q|)$ messages reaching it at step i. Clearly Q can cover points which belong to at most $n\rho|Q| + 1 \leq n(\rho+1)|Q|$ different servers, therefore $L_0^Q \leq n(\rho+1)|Q|$.

[4]It is not necessary that the messages in the i^{th} step indeed reach the segment together. There is no implied assumption of synchrony.

For a message to reach Q in the i^{th} step, it must be that on step $i-1$ the message was in an interval of length $2|Q|$ and then moved to Q. Call this interval Q'.

$$\mathbb{E}\left[L_i^Q \mid L_{i-1}^{Q'}\right] = \frac{1}{2}L_{i-1}^{Q'}$$
$$\mathbb{E}\left[L_i^Q\right] = \frac{1}{2}\mathbb{E}\left[L_{i-1}^{Q'}\right]$$

The induction hypothesis states that

$$\frac{1}{2}\mathbb{E}(L_{i-1}^{Q'}) \leq \frac{1}{2}n(\rho+1)|Q'| = n(\rho+1)|Q|.$$

There are at most $\log n + \log\rho$ steps in the first phase, and $|s(V)| \leq \frac{\rho}{n}$. Summing over i yields that the expected number of lookups that pass through V in the first phase is at most $(\log n + \log\rho)(\rho+1)\rho$, which is $O(\log n)$ when ρ is a constant.

The second phase of the routing is similar to the first phase but in reverse order. When paths are seen as going from the target towards the middle random point, then each path is a random walk determined by the random choices of the routing algorithm. Thus the analysis of the first phase holds for the second phase as well. We conclude that if $\vec{x}$ is smooth, for each server it holds that the expected number of messages it handles is $O(\log n)$.

Next we prove that w.h.p the actual number of lookups that pass through V is indeed $O(\log n)$. Let p_i $(i = 1 \ldots n)$ indicate the probability that message i passes through $s(V)$ during the first phase of the lookup. By the previous claim we know that $\sum_{i=1}^{n} p_i \in O(\log n)$. The random choices that determine the paths that messages take are independent from one another. Standard use of Chernoff's inequality yields:

$$\Pr\left[\sum p_i > \Theta((1+\epsilon)\log n)\right] \leq n^{-\Theta(\epsilon^2)}$$

Choosing ϵ large enough allows us to use the union bound over all servers thus proving the bound for the first phase. As before the analysis of the second phase is similar to that of the first. □

In reality, permutation routing is not likely to occur. Rather, servers initiate lookups for files, and the hash function which maps filenames to points should spread them evenly along I. A family of hash functions $\mathcal{H}$ is said to be *k-wise independent* if when $h \in \mathcal{H}$ is chosen randomly it holds that for any data items $m_1 \neq m_2 \neq \ldots \neq m_k$ the random variables $h(m_1), h(m_2), \ldots, h(m_k)$ are independent and uniformly distributed in I.

Assume that each server initiates one lookup. Denote by m_i the data item searched by server V_i; i.e. server V_i seeks for the server covering the point $h(m_i)$. We also assume that $m_i \neq m_j$ for all $i \neq j$.[5]

Theorem 2.11. *Given that $G_{\vec{x}}$ is smooth and h is $\log n$-wise independent, for every permutation τ it holds that when routing τ with high probability each server participates in the routing of $O(\log n)$ messages, where the probability is taken over the choice of the hash function and the random choices of the routing algorithm.*

Proof. The proof follows the same lines as the proof of Theorem 2.10. The only difference is that in the second phase it is not true that the paths messages take are independent, but rather are $\log n$-wise independent. Fix some server V and let X_i denote the event that message i was handled by V in the second phase of the routing. The analysis of Theorem 2.10 holds in this case as well so we know $E[\sum X_i]$ is $O(\log n)$ and that the X_is are $(\log n)$-wise independent. We use Theorem 4.21 in [10] which is a high moment version of Chebyshev's inequality.

[5]Section 3 deals with the case where the same item is queried by multiple servers.

Claim 2.12. *Let $X_1, X_2, \ldots, X_n$ be random variables with $0 \le X_i \le 1$ and $\mathbb{E}[X_i] = p_i$ for each $i \in [n]$. Let $X := \sum_i X_i$ and set $\mu := E[X]$ and $p := \mu/n$. For any $\delta > 0$ define $k_* := \lceil \mu\delta/(1-p) \rceil$. Then, if $X_1, \ldots, X_n$ are $k-$wise independent for $k > k_*$,*

$$\Pr[X \ge \mu(1+\delta)] \le \left(\frac{en}{\mu(1+\delta)}\right)^{k_*} \cdot p^k$$

In our case $\mu \in O(\log n)$ and $p \in O(\log n/n)$, so if $\delta = 2$ we have that $k_* \in O(\log n)$. Plugging it in it follows that there are constants c_1, c_2 such that

$$\Pr[X \ge 2\mu] \le c_1^{k_*} \cdot \left(\frac{c_2 \log n}{n}\right)^{k-k_*}$$

Now if $k - k_* = \log n$ the theorem follows. □

Theorem 2.11 demonstrates that in some sense the Distance Halving Lookup is good in a worst case scenario. An adversary may choose for each V_i its appropriate m_i (as long as the adversary is oblivious of h). It is worth noting a few facts:

- The routing is oblivious routing in the sense that the path a message passes does not does not depend on the paths of other messages. A subtle issue is that the path choice is not only a function of the destination but also depends on the numeric value of x_i.
- The routing algorithm is sensitive to small perturbations in the numerical value of the parameters. It is important to be precise enough, and to allocate enough bits for the variables. The edges of the continuous graph are tantamount to shuffling bits to the left. Since the length of all paths is bounded by $4 \log n$ it is enough to allocate $4 \log n$ bits for each variable.

2.3 Degree-Path Length Optimality

In this section we describe an emulation of the De-Bruijn graph with degree larger than 3.

Definition 4. *The r-dimensional De-Bruijn graph of degree Δ consists of Δ^r nodes and Δ^{r+1} directed edges. Each node corresponding to an $r-$bit string over the alphabet $\{0, 1, \ldots, \Delta - 1\}$. There is a directed edge from each node $u_1 u_2 \cdots u_r$ to nodes $u_2 \cdots u_r v$ for every $v \in \{0, 1, \ldots, \Delta - 1\}$.*

Every $\Delta-$regular graph has a diameter of at least $\log_\Delta n$. The De-Bruijn graph meets this lower bound: an n node De-Bruijn graph with degree Δ has a diameter of $\log_\Delta n$. Emulating this graph may reduce the lookup length and the congestion while increasing the degree.

For any $\Delta \ge 2$ construct a continuous graph with the following edges: $f_i(y) = \frac{y}{\Delta} + \frac{i}{\Delta}$ $(i = 0, 1, \ldots, \Delta - 1)$. Now both routing schemes suggested before could be applied in this case as well: Claim 2.3 translates to be $d(f_i(y), f_i(z)) = \frac{1}{\Delta} d(y, z)$ and Claim 2.4 translates to be $d(y, \sigma_t(z)) \le \Delta^{-t}$. Therefore:

Theorem 2.13. *A* smooth *discretization of the continuous graph results with a graph of degree $\Theta(\Delta)$ and with path length $\Theta(\log_\Delta n)$.*

As mentioned the diameter of any Δ-regular graph is at least $\log_\Delta n$, so for every Δ the construction has optimal path length with respect to the degree (up to constants). Two interesting options are setting $\Delta = \log n$ or $\Delta = n^\epsilon$ (for some constant ϵ), as the first results in a lookup length of $\frac{\log n}{\log \log n}$, and the second in a lookup length of $O(1)$. It is worth noting that the analysis of the previous section shows that for each choice of Δ, if the points are smooth the congestion is $\Theta(\frac{\log_\Delta n}{n})$. Thus, the increase of the degree decreases the congestion as well.

3 Dynamic Caching - Relieving Hot Spots

In this section we discuss a protocol that eliminates the occurrence of hot spots in the network. A *hot spot* occurs whenever a data item is requested simultaneously by a large number of clients - an event that happens quite often in today's internet. A highly popular data item might not only cause the server holding it to be swamped, but might also cause a bottleneck at and around its location. In order to avoid the congestion caused by a hotspot it is necessary to replicate the popular data item to other servers (i.e. caching), such that the load of handling all requests is distributed between a large number of servers. Relieving hot spots was pointed out as one of the main open problems regarding the design of DHT's by Ratnasamy *et al.*[42]. To the best of our knowledge ours is the first protocol that resolves this problem, at least in the sense of providing a provable guarantee. A detailed comparison with previous work is provided bellow. A dynamic caching protocol should satisfy four properties:

1. *Prevent Swamping:* Each server should handle as few messages as possible. This is the ultimate goal of the protocol, and should hold for every possible set of requests.

2. *Keep the Caches small:* Each server has a cache in which it stores data items. A trivial, yet inefficient, caching protocol would be to store all data items in all nodes. Such a solution of course prevents swamping, but would also have horrific performance in terms of memory use. Our goal is to keep the cache of each server as small as possible.

3. *Reduce latency:* The caching protocol may cause some delays in obtaining the desired data item. Our goal is to reduce this delay to a minimum.

4. *Keep update time low:* Content may change over time, a caching protocol must be able to accommodate efficiently changes in the cached data item itself.

Previous Work

Various caching techniques were suggested in the literature, which operate under various distributed models (e.g. [39],[19],[8],[7]). Ranade [40] was the first to deal with the problem of hot-spots in routing (for simulating PRAMs) and showed that in the Butterfly network, which is closely related to the De-Bruijn network, it is possible to prevent hot-spots by combining on-the-fly message requests along the paths to the destination. Combining messages is not an appropriate technique for an asynchronous and dynamic environment. Our approach does not combine messages but rather uses explicit caches. Chankhunthod *et al.*[7] suggested that caches be arranged as trees, where a request for the data item arrives at a random leaf of the tree and if possible handled there. If a cache does not hold the data item (and neither does its sibling) it passes the request to its parent in the cache tree until finally the message is forwarded to the root which must hold the item. The advantage of the cache tree is that requests are divided more or less evenly between the leaves and each cache receives requests from its children only, thus no cache is being swamped. Karger *et al.*[19] enhanced the idea and suggested that each data item have a different random tree of caches, thus better dividing the load. They used hash functions in order to sample a cache tree for each data item, and managed to prove that a single popular data item will not cause a hot spot with high probability.

In the context of a P2P overlay network where each server serves both as a cache and as a client, the suggestion of Karger *et al.*[19] has the following disadvantage. The random cache trees associated with each data item are not supported by the overlay network. Thus, when a cache needs to forward a request to its parent it must either use a DHT lookup or maintain a separate overlay network for this use only. The first alternative will cause a meaningful slowdown and has a high communication complexity. The second alternative requires a new link established for every data item and thus would dramatically increase

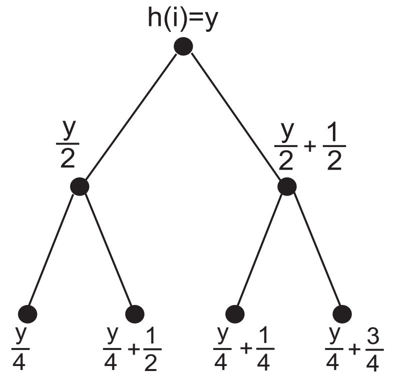

Figure 2: The first 2 layers of the path tree.

the linkage of each server and the maintenance cost associated with it. We also note that the additional messages that run in the system due to the caching protocol may cause congestion in their own sake.

Our Contribution

Our suggestion is similar in spirit to that of Consistent Hashing [19]. We also construct a random tree of servers for each data item and forward a message from a random leaf towards the root. Our scheme differs in one important point: we use the Distance Halving overlay network as the overlay network of the caching protocol as well; i.e. we couple the cache trees with the Distance Halving graph. The Distance Halving routing algorithm ensures that requests arrive at a random leaf of the cache tree. Thus the caching protocol requires no extra connections and imposes no extra delay. As a result we are able to show that any set of requests (for possibly many data items) do not swamp servers and do not cause congestion.

Given a 'batch' of n requests for data items that arrive in the network[6], the main achievement of this section is to show a dynamic caching protocol with the following properties:

- *Swamp Prevention:* Each server's cache is hit $O(1)$ times on average and $O(\log^2 n)$ with high probability. The total number of messages passed through each server (by the routing scheme and by the caching scheme) is $O(\log^2 n)$ w.h.p. This property is proven in Theorem 3.8.
- *Small Caches:* With high probability each server stores at most $O(\log n)$ data items in its cache. There are at most $O(\frac{n}{\log n})$ new copies of data items stored in caches throughout the network. This property is proven in Theorem 3.8.
- *No Caching Latency:* The caching protocol causes *no* extra delay.
- *Quick Content Update:* Changes in the data item are updated in $O(\log n)$ time.

3.1 The Protocol

As usual we first describe the protocol in the continuous graph. Denote by $\mathcal{D}$ the set of data items. Let $i \in \mathcal{D}$ be a popular data item and denote y to be $h(i)$, so the server which covers y also holds a copy of i.

Definition 5. *The path tree rooted at y is a subgraph of the continuous graph G_c. The tree is created by assigning y as the root, and each node z in the tree is the parent of $\ell(z), r(z)$.*

[6]Since n denotes the number of servers in the network, we assume that each server issues one request.

The first two layers of the path tree are drawn in Figure 2. The key observation is that if the Distance Halving routing algorithm (of Section 2.2.2) is used , then every request for i reaches y via a *random path* in the path tree; i.e. the probability that a message reaches y via the point $\frac{y}{2}$ is half, and so on. This observation suggests that it is wise to replicate data item i from the root of the tree downward, thus creating a cache tree (a-la [7],[19]). If the data item is copied into the nodes of some layer, then the randomness of the routing protocol ensures that requests are divided evenly (more or less) among the nodes. In other words, the continuous graph may serve as a random *cache tree* where the nodes of the path tree hold the data item. We call a node that holds a copy of the data item i an *active node*. The tree which consists of all the active nodes is the *active tree*.

In order to deal with a dynamic caching setting formally, we need to define the dynamics of the requests. Assume all servers of the system count[7] the same time epoch. Each server decides upon some number c that is a parameter of the protocol and serves as a threshold. Typically c would be in the order of $\log n$ and may be updated over time. It is not necessary that all servers choose the exact same parameter, yet for sake of convenience we assume that c is a global parameter known to all servers. The term *node* is used only in the context of the path tree, while the term *server* refers to the participants of the network.

Continuous Hot Spots Protocol:

1. Each leaf of the active tree holds a counter which indicates the number of times of the data item was requested during an epoch. Once a data item is requested more than c times, the leaf replicates the data item into both its children, effectively blocking itself from handling subsequent requests.

2. If z is the parent of two leaves of the active tree, then at the end of an epoch z performs the following procedure: It checks how many times i was supplied by its children during the epoch. If i was supplied less than c times by both its children, then both children may delete the item i and cease to be active. As a consequence the point z becomes a leaf of the active tree.

3. Step 2 repeats itself recursively, in the same epoch, collapsing the active tree if there are no requests.

In practice it may be beneficial to set a *different* threshold in Step (1) and Step (2). This adds stability to the active tree when the rate of requests is close to the threshold. It also may be more efficient to modify Step (1) such that the data item is initially copied into *one* child, and after another c requests into the second. While both modifications may increase efficiency slightly, they also complicate the analysis.

Presentation In Section 3.2 we analyze the protocol on the continuous graph, i.e. on the active tree. In Section 3.3 we analyze the case of a single hot spot. In Section 3.4 we analyze the more general case, in which multiple hot spots are formed.

3.2 The Active Tree

Denote by q_i the number of times i is requested during an epoch. Each node of the active tree handles at most c requests. Two siblings are deleted if they handled less than c requests each, therefore:

Observation 3.1. *For every initial active tree, by the end of the epoch the active tree contains at most $\frac{4q_i}{c}$ nodes. Therefore, the total size of caches in the network is at most $\frac{4q_i}{c}$.*

Observation 3.2. *The distance between two points in the j^{th} layer of a path tree is at least 2^{-j}.*

[7] We don't assume that the system is synchronized, this assumption is for convenience and does not play a major role.

Next we show how the active tree grows and shrinks as a function of q_i. The threshold c is assumed to be at least $\log n$.

Lemma 3.3. *If $c \geq \log n$ then with probability at least $1 - \frac{1}{n}$, the lowest point in the active tree at the end of the epoch is at layer at most $\log(\frac{q}{c}) + O(1)$, where the probability is taken over the random decisions of the Distance Halving routing scheme.*

Proof. If for some layer j of the tree it holds that each node in the layer receives less than c messages, then clearly the tree cannot exceed layer j. Consider layer $j = \log(\frac{q}{c}) + t$ of the tree, for some constant t. There are $\frac{q}{c}2^t$ nodes in layer j. The randomness of the routing algorithm causes each message to reach the target independently through a random node in the layer. A standard balls and bins argument shows that with high probability each node in level j received at most c messages: The probability a message reaches a fixed node in level j is 2^{-j}. The total number of messages reaching the node is distributed according to the Binomial distribution with expectation $2^{-j}q = 2^{-t}c$. Chernoff's bound (e.g. [3]) states that for a binomial variable X it holds that $\Pr[|X - \mu(X)| \geq \epsilon\mu(X)] \leq 2e^{-\delta\epsilon\log\epsilon\mu(X)}$ where $\epsilon > 10$ and δ is a constant independent of ϵ, n. Substituting $\mu(X)$ for $2^{-t}c$ and ϵ for 2^t we get that the probability the number of messages handled by a node exceeds c is at most $e^{-\Theta(tc)}$. If t is large enough and c is $\Omega(\log n)$, then with probability at most $\frac{1}{n^2}$ the node receives at most c messages. Union bounding over the 2^j nodes in the layer yields that with probability at least $1 - \frac{1}{n}$ all nodes of layer j receives less than c messages. ☐

Lemma 3.4. *The load on each active node is bounded as follows:*

1. *The cache at each node is hit at most c times.*
2. *Given that c is $\Omega(\log n)$, w.h.p each active node passes at most $O(\log n)$ messages up to its parent where the probability is taken over the random choices of the routing algorithm.*

Proof. Once a cache is hit more than c times in the same epoch the node replicates the data item to its children, effectively blocking it from being hit again in this epoch and the next. To prove the second claim consider a node at level $j + 1$ and denote by X the number of messages it passed on to its parent. There are at most 2^j active nodes in the first j levels of the tree, therefore at most $c2^j$ requests were passed to the first j levels of the tree by the 2^{j+1} nodes in level $j + 1$, so X has the binomial distribution with expectation at most $O(c/2)$. Now the analysis is similar to that of Lemma 3.3. Chernoff's bound states that $\Pr[|X - \frac{c}{2}| \geq \epsilon\frac{c}{2}] \leq e^{-\Theta(\epsilon\log\epsilon\frac{c}{2})}$. Since c is $\Omega(\log n)$ we have that for ϵ large enough, with probability $1 - \frac{1}{n}$ the number of messages passed on by each of the nodes is $O(\log n)$. ☐

3.3 Analysis of a Single Hotspot

In the discrete protocol (as usual) server V emulates all the points in $s(V)$. In Figure 3 it is demonstrated how the nodes of the active tree are mapped to the servers.

Lemma 3.5. *Server V covers with high probability $O(\log(\frac{q}{c}) + \frac{q}{c}|s(V)|)$ nodes of the active tree, where the probability is over the random choices of the routing algorithm.*

Proof. Observation 3.2 implies that server V covers at most $\lceil|s(V)|2^j\rceil$ points from layer j. It may be that V covers one point from each layer of the tree. If for instance V covers the point 0 and the root of the tree, then it also covers all the points in the left branch of the tree. Summing over the levels of the tree Lemma 3.3 implies that the number of times V served as a cache is w.h.p.

$$\sum_{j=0}^{\log\frac{q}{c}+O(1)} \left(\lceil|s(V)|2^j\rceil\right) \leq \sum_{j=0}^{\log\frac{q}{c}+O(1)} |s(V)|2^j + 1 = O(\log(\tfrac{q}{c}) + \frac{q}{c}|s(V)|)$$

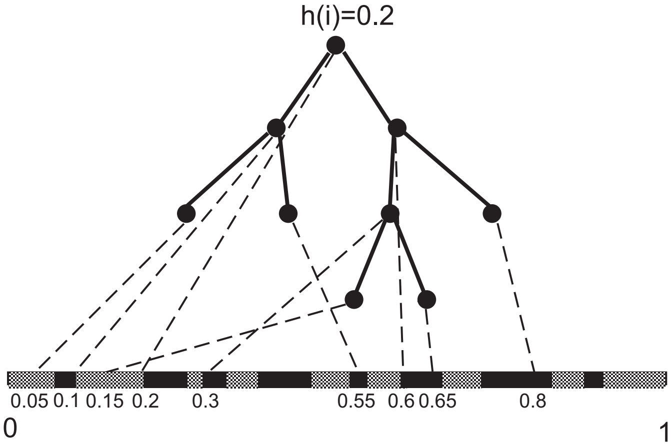

Figure 3: The mapping of an active tree to the servers. The bold edges represent the active tree. The dashed edges represent the mapping. The $[0,1)$ interval is divided to segments which represents the servers.

□

Lemma 3.4 bounds the load of each node in the active tree, so Lemma 3.5 combined with Lemma 3.4 immediately implies the following:

Theorem 3.6. *The load on each server is bounded as follows:*

1. *The cache at each server is hit at most $O(\log(\frac{q}{c})c + q|s(V)|)$ times w.h.p.*
2. *Given that c is $\Omega(\log n)$, w.h.p each server passes at most $O(\log n)$ messages where the probability is taken over the random choices of the routing algorithm.*

The Theorem states that the number of times V served as a cache is $O(c\log(\frac{q}{c}) + q|s(V)|)$. Given that c is $\Theta(\log n)$ and $q \leq n$ (each server may issue at most one request per time epoch then), and $|s(V)| \leq \frac{\log^2 n}{n}$ (very reasonable) then the bound translates to $O(\log^2 n)$. Note that Lemma 3.5 uses only the randomness of the routing protocol and does not assume that the hash function h has any specific properties. It holds even if an adversary is allowed to choose $h(i)$.

In the following we make a mild assumption over the hash function. We assume the hash function h is randomly chosen from a family of hash functions $\mathcal{H}$. The family $\mathcal{H}$ has the property that for every i it holds that $h(i)$ is uniformly distributed in I. This property is sometimes called *one-wise independence*. We stress that this is a very weak requirement; for instance, the common notion of a pairwise independent family of hash functions satisfies this requirement. As before we need to count the number of active nodes covered by V. Denote by B_v the number of nodes covered by V. Now B_v depends both on the randomness of the routing algorithm and on the randomness of the hash function.

Lemma 3.7. *If $h(i)$ is uniformly distributed in I and $|s(V)|$ is $O(\frac{\log n}{n})$, then for every $t > 0$:*

$$\Pr[B_v \geq t] \text{ is } O\left(|s(V)| \cdot \frac{q}{c} \cdot 2^{-\Theta(t)}\right)$$

where the probability is over the choice of the hash function h and the routing algorithm.

Proof. Consider level j of the path tree. According to Observation 3.2, $s(V)$ covers at most $\lceil |s(V)|2^j \rceil$ active nodes. Lemma 3.3 bounds the depth of the active tree by $\log(\frac{q}{c}) + O(1)$, and $|s(V)|$ is bounded by $\log n$. We conclude V covers at most $O(1)$ nodes from each level of the active tree. Therefore if $B_v \geq t$ then V covers at least one node from the first $\log(\frac{q}{c}) - \Theta(t) + O(1)$ levels of the tree.

We proceed by calculating the probability V covers at least one node from level j. Every specific level j node of the path tree is uniformly distributed (by the hash function) in a segment of length 2^{-j}. For instance the leftmost node of level j is uniformly distributed in the segment $[0, 2^{-j}]$ and so on. Therefore the expected number of nodes V covers in level j is $O(|s(V)|2^j)$. We conclude that the probability V covers at least one node in level j is bounded by $O(|s(V)|2^j)$.

Now we have: $\Pr[B_v \geq t] \leq \Pr[V$ covers at least one point in the first $\log(\frac{q}{c}) - \Theta(t) + O(1)$ levels$]$ which is at most

$$\sum_{j=0}^{\log(\frac{q}{c})-\Theta(t)+O(1)} |s(V)|2^j = O\left(|s(V)| \cdot \frac{q}{c} \cdot 2^{-\Theta(t)}\right).$$

□

By linearity of expectation, the expected number of active nodes V covers is $O(|s(V)|\frac{q}{c})$. When $|s(V)|$ is $\Theta(\frac{1}{n})$ (the graph is smooth) and $q \leq n$ the bound translates to $O(\frac{1}{c})$. Therefore the expected number of requests handled by V is $O(|s(V)|q)$, which is $O(1)$ when the graph is smooth.

3.4 Multiple Hotspots

In this Section we analyze the more general case where there is an arbitrary set of demands for n data items. We make the following assumptions: the threshold is set $c = \Theta(\log n)$, the demand for data item i is q_i such that $\sum_i q_i = n$, the hash function h which maps the data items into I is randomly chosen from a $\log n$-wise independent family and the graph is smooth.

The following is the main result of this Section:

Theorem 3.8. *If h is $k-$wise independent where $k \geq \log n$, then for all sequences of n requests:*

- (i) - *Given that $|s(V)| \leq \frac{\log n}{n}$ for each server V, with probability at least $1 - \frac{1}{n}$ the maximum number of items stored at any of the caches is $O(\log n)$, where the probability is taken over the random choices of the routing algorithm and the hash function.*
- (ii) - *For each server V the expected number of times V supplies a data item is $O(|s(V)|n)$. Given that $|s(V)| \leq \frac{\log n}{n}$, with probability $1 - \frac{1}{n}$, each server supplies data items at most $O(\log^2 n)$ times.*

Proof. Fix a server V. Denote by p_i the probability V supplies data item i. According to Lemma 3.7 we have that p_i is $O(\frac{q_i|s(V)|}{c})$. Denote by L_v the number of data items supplied by V. We have

$$\mathbb{E}[L_v] = \sum_i p_i = O\left(\sum_i \frac{q_i|s(V)|}{c}\right) = O\left(\frac{n|s(V)|}{c}\right).$$

Recall that $c \geq \log n$ and $|s(V)| \leq \frac{\log n}{n}$ so $\mathbb{E}[L_v]$ is $O(1)$. The random variable L_v is the sum of $(\log n)$-wise independent Bernoulli variables. Using Claim 2.12 we have that:

$$\Pr[L_v \geq \delta \log n] \leq \Pr\left[|L_v - \mathbb{E}[L_v]| \geq \delta' \log n \cdot \mathbb{E}[L_v]\right] \leq O\left((\delta^2 \log n)^{-\frac{1}{2}\log n}\right)$$

where δ' is some constant depending upon δ. The proof of part (i) is completed by setting the constant δ large enough so that $\Pr[L_v \geq \delta \log n] \leq \frac{1}{n^2}$ and union bounding the probabilities for the n servers.

Part (ii) is composed of two statements. The first follows directly from Lemma 3.7 which states that the expected number of times V handles the i'th data item is $\mathbb{E}[q_i|s(V)|]$. Therefore the expected number of times V supplies a data item is $O(\sum_i q_i|s(V)|) = O(|s(V)|n)$.

For the second part let B_i denote the random variable counting the number of active nodes V covers from the i^{th} active tree. Lemma 3.7 implies that there is a constant a such that w.h.p $B_i \leq a \log n$. Furthermore $\mathbb{E}[\sum B_i]$ is $O(\sum \frac{|s(V)|q_i}{c})$ which is $O(1)$. Define $X_i := B_i/a \log n$ and $X := \sum X_i$. Now, w.h.p $0 \leq X_i \leq 1$ and $\mathbb{E}[X] \in O(1/\log n)$. It holds that for every α there is an α' such that $\Pr\left[\sum B_i \geq \alpha \log n\right] \leq \Pr[X \geq \alpha']$. Now Claim 2.12 implies that that there is a constant α' such that $\Pr[X \geq \alpha'] \leq \frac{1}{n^2}$ which concludes the proof of Theorem 3.8. ☐

Content Update The tree-like structure of the cache means that the popular data item may be changed or altered efficiently by propagating the update from the owner of the data (the root) along the active tree. As a consequence, an update of all the caches in which the data item is stored takes $O(\log \frac{q}{c}) \leq O(\log n)$ time and $O(\log \frac{q}{c}) \leq O(\log n)$ messages.

Summary We have shown that our caching scheme satisfies the properties required: Hot spots are eliminated with high probability for every set of requests, caches of servers are small, and most strikingly the caching scheme causes *no extra delay* for obtaining the data item. All done in a dynamic and scalable manner. The caching scheme uses the fact that in the Distance Halving *continuous* graph, every node is the root of an infinite binary tree. The same property was also used by Nadav and Naor [34] in order to build fault tolerant storage systems.

4 Load Balancing the ID's - Achieving Smoothness

We have seen that the smoothness of the decomposition of I determines the efficiency of many of our protocols. In this section we suggest various distributed algorithms in which a server can choose its i.d. (i.e. perform the first step of Algorithm Join in Section 2). The goal is that all servers choose their i.d. such that I is smoothly divided between them. An algorithm for choosing an i.d. should be as local and efficient as possible. Ideally the smoothness should be maintained even under worst case conditions where an adversary chooses which and when servers join and leave the system. In practice most of the algorithms we show assume that some properties are behaving randomly. The problem of achieving smoothness was also addressed in [1],[30],[20]. In the following we always assume that I is continuous. All bounds remain correct even if points are perturbed by a polynomially small values, therefore a allocating $3 \log n$ bits per variable is enough in order for the theorems to be correct in the discrete case.

A straightforward algorithm, that was also suggested by previous constructions (e.g. [45],[29]) is letting each server choose its x-value by sampling randomly and uniformly a point in I:

Algorithm Single Choice for server V

1. Choose $V.ID$ in $[0, 1)$ uniformly at random.

The following lemma is proven in [29].

Lemma 4.1. *After inserting n random points the length of the longest segment is w.h.p $\Theta(\frac{\log n}{n})$. With high probability there is no segment which is shorter than $\Theta(\frac{1}{n^2})$*

An important property of the Single Choice Algorithm is that Lemma 4.1 holds even if a random subset of the points is deleted, i.e; a random subset of the servers quit the system. In the Single Choice Algorithm segments remain small w.h.p, and in fact a careful look at our proofs shows that this is enough, i.e. using this scheme would only cause a logarithmic blowup in parameters. The main drawback of the scheme is that it may allow some segments to be very small - of length $O(\frac{1}{n^2})$, which means that the servers covering them hardly share any of the load. A slight improvement is the following.

Improved Single Choice Algorithm for server V

1. Choose a point $z \in [0, 1)$ uniformly at random.
2. Lookup z and find the bounderies of the segment of the server which currently covers z.
3. Set $V.ID$ to be *middle* point in that segment.

Lemma 4.2. *In the improved single join algorithm, the shortest segment is of length $\Theta(\frac{1}{n \log n})$ w.h.p and the longest segment remains $O(\frac{\log n}{n})$.*

Proof. Simulate the process of choosing the $x_1, x_2, \ldots, x_n$ points by growing a random binary tree in the following way: Let $z_1, z_2, \ldots, z_n$ denote the n random points chosen at Step (1) of the algorithm, where z_i is the point chosen by the ith server. The point x_1 always takes the value $\frac{1}{2}$, which corresponds to the root of the tree. The point x_2 takes a value of $\frac{1}{4}$ with probability $\frac{1}{2}$ (if $0 \leq z_2 < \frac{1}{2}$), in which case we add a left child to the root. The point x_2 takes a value of $\frac{3}{4}$ with probability $\frac{1}{2}$, in which case we add a right child to the root, and so on. Each time a point is added, it randomly selects a path from the root until it reaches a leaf (the path is determined by the binary representation of z), and then becomes either the left or the right child of that leaf. Now consider the layer of the tree in which there are $\Theta(n \log n)$ nodes. For each node of the tree, with high probability there is at most one z-point which reached it (balls and bins). It immediately follows that the smallest segment is of length $\Omega(\frac{1}{n \log n})$ with high probability. Consider the layer of the tree which has $\Theta(n/\log n)$ nodes. With high probability each one of them were hit by at least one of the z-points. It follows that the largest segment is of size $O(\frac{\log n}{n})$. □

The idea of the following algorithm (following the spirit of the two choice paradigm, see [33] for a survey), is to let a joining server choose *many* locations and set its ID to be the best location found.

Multiple Choice Algorithm

1. Estimate $\log n$.
2. Sample $t \log n$ random points from I, when t is some constant to be determined later.
3. Check all segments containing those points. Let s be the longest of these segments. Set ID to be the middle point of s.

Estimating $\log n$ is a simple task and can be done in various ways which are discussed in Section 6. For convenience we assume the estimation is completely accurate. The problems caused by the inaccuracy of the estimation could be overcome by slightly increasing the parameter t.

Lemma 4.3. *If $t \geq 2$, after n points are inserted, the length of the shortest segment is at least $\frac{1}{4n}$ with probability at least $1 - \frac{1}{n}$.*

Proof. A segment of length $\frac{1}{4n}$ could be created only if all the $t \log n$ samples fell in a segment of length at most $\frac{1}{2n}$. The probability a random point in I falls in some segment of length $\frac{1}{2n}$ is at most $\frac{1}{2}$. The probability all the $t \log n$ fell in these segments is at most $2^{-t \log n} = \frac{1}{n^t}$. Now, since $t \geq 2$ a union bound over the error probabilities of all the points proves the lemma. □

In the following we show that the Multiple Choice Algorithm has a self correction property. Even when an adversary controls the initial state of the system, after the injection of n more points, with high probability no segment is big.

Theorem 4.4. *For any $m > 0$ and **any** configuration of m points in I, after inserting n more points (servers), the largest segment is of size at most $O(\frac{1}{n})$, with probability $1 - \frac{1}{n}$.*

Proof. First we prove the following lemma:

Lemma 4.5. *Assume the longest segment is of length $\frac{c}{n}$, then for sufficiently large t, the expected number of new points needed to be inserted in order to reduce the longest segment to a length of $\frac{c}{2n}$ is at most $\frac{3n}{c}$ points. Inserting $\frac{20n}{c}$ new points suffices with probability $1 - \frac{1}{n^2}$. The probability is taken over the random choices of the algorithm.*

Proof of Lemma 4.5. Denote by k the number of segments of length between $\frac{c}{2n}$ and $\frac{c}{n}$. There are at most $\frac{2n}{c}$ such long segments. Our goal is to calculate how many points we should insert into the system until all these k segments are split. Denote by X_i the random Geometric variable counting the number of new insertions until the next large segment is split when there are i long segments remaining. The total number of inserted new points is therefore $\sum_{i=1}^{k} X_i$. We denote by p_i the parameter of X_i. The probability that a uniformly sampled point in I falls within a long segment is at least $\frac{ic}{2n}$, therefore we have

$$p_i \geq 1 - \left(1 - \frac{ic}{2n}\right)^{t \log n} \tag{4}$$

First we deal with the case that $i \leq \frac{2n}{ct \log n}$:

$$\left(1 - \frac{ic}{2n}\right)^{t \log n} \leq e^{-\frac{ict \log n}{2n}} \leq 1 - \frac{ict \log n}{4n}$$

where the second inequality follows since $\frac{ict \log n}{2n} \leq 1$. we conclude that for $i \leq \frac{2n}{ct \log n}$ it holds that

$$\mathbb{E}[X_i] \leq \frac{1}{1 - (1 - \frac{ic}{2n})^{t \log n}} \leq \frac{4n}{ict \log n}$$
$$\sum_{i \leq \frac{2n}{ct \log n}} \mathbb{E}[X_i] \leq \frac{4n}{ct \log n} \cdot \sum \frac{1}{i} \leq \frac{4n}{ct}.$$

Next we show that

$$\Pr[\sum_{i \leq \frac{2n}{ct \log n}} X_i \geq \frac{20n}{ct}] \leq \frac{1}{n^2}$$

Consider only the first $\log n$ variables. Clearly $\mathbb{E}[\sum_{i \leq \log n} X_i] \leq \frac{4n}{ct}$. We bound $\Pr[\sum_{i \leq \log n} X_i \geq \frac{20n}{ct}]$ by tossing $\frac{20n}{ct}$ biased coins, each coin turns Head with probability $\frac{4n}{ct \log n}$. Denote by Y the number of successes, $\mathbb{E}[Y] = 5 \log n$. Now:

$$\Pr[\sum_{i \leq \log n} X_i \geq \frac{20n}{ct}] \leq \Pr[Y \leq \log n] \leq \frac{1}{n^2}.$$

The same argument for the next $\log n$ variables shows that

$$\Pr[\sum_{\log n < i \leq 2\log n} X_i \geq \frac{20n}{ct\log n}] \leq \frac{1}{n^2}$$

All in all, with high probability the total number of new points is at most $\frac{20n}{ct}(1 + \frac{1}{\log n} + \frac{1}{\log^2 n} + ...) = \frac{20n}{ct}(1 + \frac{1}{\log n - 1})$.

Now deal with the case $\frac{2n}{ct} \geq i > \frac{2n}{ct\log n}$. In this case $\mathbb{E}[X_i] \leq \left(1 - (1 - \frac{ic}{2n})^{t\log n}\right)^{-1} \leq 2$ and therefore the expected total contribution of these variables is at most $\frac{4n}{ct}$. As before, the probability more than $\frac{20n}{ct}$ points are needed is at most $\frac{1}{n^2}$. Finally when $\frac{2n}{c} \geq i > \frac{2n}{ct}$ it holds that $\mathbb{E}[X_i] \leq \frac{n}{n-1}$ and the total contribution of these variables is at most $\frac{2n}{c}$. Therefore with high probability the total number of points inserted does not exceed $\frac{20n}{ct}(1 + \frac{1}{\log n - 1}) + \frac{20n}{ct} + \frac{2n}{c}$. Choosing $t = 20$ we conclude that on expectation it takes at most $\frac{3n}{c}$ insertions to split all long segments, and at most $\frac{20n}{c}$ new insertions with high probability. $\square$

The proof of Theorem 4.4 is completed by applying the lemma iteratively $\log c - 3$ times. Thus, we have that it takes on expectation n new points until the longest segment is of length at most $\frac{8}{n}$. With high probability after inserting n points the longest segment is of length $O(1/n)$. $\square$

We stress that the Multiple Choice Algorithms self corrects any initial configuration only in the sense that it reduces the size of the largest segment. Any extremely small segment in the initial configuration would remain small after the new points were inserted.

4.1 Handling Deletions

The Multiple Choice Algorithm achieves a smooth set of points in the pure Join model,when servers may join the system but not leave it. Achieving smoothness in a setting which allows deletions requires a more complicated scheme.

The most naive way to handle the deletion of a point is to assign its segment to its predecessor on the ring. It is easy to see that this naive algorithm does not suffice: Assume that there are $2n$ points spread smoothly in I, and randomly delete each one with probability $\frac{1}{2}$. With high probability there is a sequence of $\Omega(\log n)$ *consecutive* points that were deleted, creating a segment of length $\Omega(\frac{\log n}{n})$ and violating the smoothness. We conclude that some balancing mechanism must be implemented in order to maintain smoothness under deletions.

The Bucket Solution The 'bucket' solution was suggested in Viceroy [29]. The servers join the system with the Single Choice algorithm . The balance is achieved via an extra structure. We maintain a distributed coordination mechanism between contiguous chains of servers, consisting of $O(\log n)$ servers each. We call such a group of $O(\log n)$ servers a *bucket*. Inside each bucket we maintain a simple ring (which mostly overlaps the larger ring of the DH construction). The buckets are maintained such that two properties hold:

1. The size of the bucket is always $\Theta(\log n)$. When the size of a bucket exceeds $c\log n$ (for some constant c) it splits into two. When the size of a bucket shrinks bellow a threshold, it merges with a neighboring bucket. An estimation of $\log n$ is maintained within each bucket.

2. Within each bucket segments are distributed evenly, i.e. servers may change their i.d. *slightly* so that no segment is too big or too small.

The exact way in which Step (2) is performed may vary according to context. Keeping all segments within a bucket of equal length at all times may have a large overhead, as all members of the bucket update their state whenever a node joins or leaves. Therefore it makes more sense for the members of the bucket to rearrange themselves only when the smoothness within the bucket exceeds some tunable parameter.

The correctness of the bucket solution follows from the fact that w.h.p every interval of length $\frac{\log n}{n}$ contains $\Theta(\log n)$ points (balls and bins). Thus, when a segment becomes too big or too small a balancing within the bucket suffices.

4.2 Other Solutions

In [35] we describe an algorithm called the *Cyclic Scheme* which can handle adversarial deletions, albeit with low throughput. Numerous algorithms that achieve smoothness were published subsequent to [35]. Karger and Ruhl [20] suggest an algorithm that achieves smoothness in a model that also allows deletions. Upon insertion or deletion $O(\log \log n)$ nodes need to change their position. Kenthapadi and Manku [21] generalize the analysis of the Multiple Choice Algorithm to the case where some of the probes are sequential in the key space.

5 Higher Dimensions

So far we limited ourselves to the case where I is a one dimensional unit interval. Indeed, as was shown by previous constructions (such as Chord, Pastry and Viceroy) a one dimensional name space suffices for a Distributed Hash Table. In some applications it may be useful to apply the same approach to a two dimensional universe. In [37] the continuous-discrete approach was used to construct a dynamic *quorum system*. In this paper we show how to construct a network which is an expander.

In the two dimensional case the set of points cannot be associated with segments, but rather be associated with a tessellation of the plain into cells, such that each point is associated with a cell and is responsible for all keys that fall within that cell. This was done in CAN [41] where the plain was divided into rectangles. We present a simpler way to do it using the points as generators to a planar ordinary Voronoi diagram.

5.1 Dynamic Voronoi Diagrams

Definition 6 (planar ordinary Voronoi diagram)**.** *Given a finite number (at least* 2*) of distinct points in the Euclidean plane, we associate all locations in that space with the closest member(s) of the point set with respect to the Euclidean distance. The result is a tessellation of the plane into a set of regions associated with members of the point set. We call this tessellation the* planar ordinary Voronoi diagram *generated by the point set, the points are sometimes referred to as* generators *and the regions constituting the Voronoi diagram* Voronoi cells*. The dual triangulated graph is called the* Delaunay triangulation*. See Okabe et al.[38] for a thorough overview of Voronoi diagrams and their applications.*

Given an existing Voronoi diagram, the entrance of a new generator and the exit of an existing one affects only the cells adjacent to the location of the generator. Therefore a Voronoi diagram can be maintained by a distributed algorithm, in which every cell is calculated separately and *locally*. As a result the time and memory needed to compute a single Voronoi cell is $\Theta(d)$ when d is the number of neighbors the cell has; i.e. the degree of the generator in the Delaunay tessellation. It is known that d is always 6 on average (Euler's formula), but might be as high as $n-1$. In the following we set $I = [0,1) \times [0,1)$. Let $\vec{x}$ be a set of n points in I.

Definition 7. *We say that $\vec{x}$ has smoothness ρ if the following two conditions hold: (1) when dividing the rectangle to ρn rectangles of size $\frac{1}{\sqrt{\rho n}} \times \frac{1}{\sqrt{\rho n}}$, each rectangle contains at least one point from $\vec{x}$. (2) when dividing the rectangle to $\frac{n}{\rho}$ rectangles of size $\sqrt{\frac{\rho}{n}} \times \sqrt{\frac{\rho}{n}}$, each rectangle contains at most one point from $\vec{x}$.*

A simple geometric argument shows that when $\vec{x}$ is smooth, the area of each Voronoi cell generated by $\vec{x}$ is $\Theta(\frac{1}{n})$, see [37] for more details.

5.2 Constructing Expander Graphs

Expander graphs are graphs that are very 'well connected' in the sense that for every set of vertices S of size at most $\frac{1}{2}|V|$ there are at least $\alpha|S|$ vertices in $V \setminus S$ that are adjacent to some vertex in S. In this case we say the *expansion* of the graph is α. Expander graphs are probably one of the most researched structures in combinatorics. They have numerous applications in computer science. Applications in distributed computing include load balancing, fault tolerance and search through random walks (c.f. [2],[9],[27],[15],[4]).

It is well known that a random regular graph is an expander with high probability [13]. An explicit and deterministic construction for expanders was given by Margulis [31] and Gabber and Galil [14], and was later generalized by Cai [6].

The goal in this Section is to construct a P2P network which is guaranteed to be an expander. Independently from this paper, Law and Siu [25] used the fact that random graphs are expanders w.h.p to construct an expander in a P2P setting. We take a different approach, we use a two dimensional name space and the continuous-discrete approach in order to emulate the Margulis, Gabber-Galil expander. The main advantage of our approach is that in our case the expansion of the network could be verified. If the set of ID's is smooth then the network is *guaranteed* to be an expander. Gabber and Galil define a continuous graph G over I by the following two transformations: $f(x, y) = (x + y, y) \mod 1$, $g(x, y) = (x, x + y) \mod 1$. The neighbors of point $(x, y) \in I$ are $f(x, y), g(x, y), f^{-1}(x, y), g^{-1}(x, y)$. For any set $A \subseteq I$ define $\delta(A)$ to be the set of points in $I \setminus A$ which are neighbors of a point in A. Denote by $\mu(A)$ the area of the set A (its Lebesgue measure).

Theorem 5.1 ([14])**.** *For every set A of points in I such that $\mu(A) \leq \frac{1}{2}$ is well defined, it holds that $\mu(\delta(A)) \geq \frac{(2-\sqrt{3})}{2}\mu(A)$.*

Corollary 5.2. *Let $\vec{x}$ be a set of n points in I. Let $G_{\vec{x}}$ be the discretization of the Gabber-Galil continuous graph. The maximum degree of $G_{\vec{x}}$ is $\Theta(\rho)$, and the expansion of $G_{\vec{x}}$ is $\Omega(\frac{(2-\sqrt{3})}{\rho})$. So if ρ is constant $G_{\vec{x}}$ is a constant degree expander.*

Any network created by maintaining a Voronoi diagram of a smooth set of points guarantees expansion. Currently no know routing scheme is known for the general Gabber-Galil expander. A step towards a routing scheme was given by Larsen [24] which provides an algorithm for finding short routes in the Gabber-Galil expander whenever the number of nodes is a square of a prime. In this case the construction yields an efficient constant degree expanding DHT. In contrast, in the construction of Law and Siu [25], in order for the expander to be navigable the degree should be logarithmic.

5.3 Achieving Smoothness in Two Dimensions

In this section we show that a natural generalization of the Multiple Choice algorithm, achieves smoothness with high probability in the two dimension setting. We need to assume that the network supports a lookup operation for the points in I (for instance by a DHT). For a point $z \in I$ define $r(z)$ to be the rectangle containing z when I is divided to $2n$ rectangles of size $\frac{1}{\sqrt{2n}} \times \frac{1}{\sqrt{2n}}$. Let $R(z)$ be the rectangle containing z when I is divided to $n/2$ rectangles of size $\sqrt{\frac{2}{n}} \times \sqrt{\frac{2}{n}}$.

2D Multiple Choice

1. Sample $t \log n$ random points in I. Call these points $z_1, z_2, \ldots, z_{3 \log n}$.
2. For each z_i perform Lookup(z_i) and check whether $r(z_i)$ and $R(z_i)$ are empty.
3. If there exists an i such that $r(z_i)$ is empty and $R(z_i)$ is empty then set $x \leftarrow z_i$.
4. Otherwise find an i such that $r(z_i)$ is empty and set $x \leftarrow z_i$. If no such i exists then the algorithm failed, set $x \leftarrow z_1$.

We assume for convenience that the estimation of n is accurate. A multiplicative estimation of n is easily achievable and suffices.

Lemma 5.3. *After inserting n points using the 2D Multiple Choice algorithm, with probability $1 - \frac{1}{n}$ it holds that the smoothness of $\vec{x}$ is as most 2.*

Proof. First we show that w.h.p every small rectangle will contain at most one point from $\vec{x}$, i.e. Step (4) of the algorithm never fails. There are $2n$ small rectangles, at most n of them are not empty. The probability to hit a non empty rectangle is at most $\frac{1}{2}$. The probability that all $t \log n$ samples hit a non empty rectangle is at most $\frac{1}{n^t}$. Thus, the probability the algorithm failed for some point is at most $\frac{1}{n^2}$ for any $t \geq 3$.

Next we need to show that at the end of the algorithm w.h.p all the big rectangles contain at least one point. There are $n/2$ big rectangles. Assume k of them already contain a point from x. The probability a random point from I hits one of these k rectangles is $\frac{2k}{n}$. The probability that when inserting α_k points only non-empty rectangles were hit is at most $\frac{2k}{n}^{\alpha_k t \log n}$. This implies that when $\frac{3}{t \log(n/2k)}$ points are inserted, the probability of hitting non-empty rectangles only is at most $\frac{1}{n^3}$. Now for some $t = O(1)$ it holds that

$$\sum_{i=0}^{n/2-1} \frac{3}{t \log(n/2i)} \leq \frac{3}{4} n$$

which means that with probability $1 - \frac{1}{n^2}$ after inserting $\frac{3}{4}n$ points, all the big rectangles are full. The proof is completed by union bounding the error probabilities. $\square$

6 Fault Tolerant constructions

In this section we present a fault tolerant DHT. There are two common methods for modeling the occurrence of faults. The first is the random fault model, in which every server becomes faulty with some probability and independently from the other servers. The other is the worst case model in which an adversary which knows the state of the system chooses the faulty subset of servers. There are several models that describe the behavior of faulty servers. One of them is the fail-stop model in which a faulty server stops sending messages altogether. Another is a false message injection model in which a faulty server may produce arbitrary false versions of the data item requested, but otherwise behaves correctly. For instance, the false message injection model does not allow a server to send inconsistent messages regarding the state of the network itself, and in particular a faulty server *joins* the network according to protocol. A third model is the Byzantine model in which there are no restrictions over the behavior of faulty servers.

Our construction is based on the DH *continuous* graph. It differs from the construction of Section 2, only in the discretization, by letting the segments of the vertices *overlap*. In the random fault model, if we want all servers to be able to access all the data items then it is necessary that the degree be at least $\Omega(\log n)$ and that every data item is stored by at least $\Omega(\log n)$ servers. Otherwise with high probability there would be non faulty servers disconnected from the system, and data items completely irretrievable. Indeed our construction has logarithmic degree. We show two routing algorithms. The first has time and message complexity of $O(\log n)$. It guarantees that in the random fail-stop model w.h.p *all* servers can locate *all* data items. The second routing algorithm guarantees the same but under the random false message injection model. This algorithm has running time (parallel) of $O(\log n)$ and message complexity of $O(\log^3 n)$.

6.1 Related Work

Several peer-to-peer systems are known to be robust under random deletions ([48], [45], [41]). Stoica *et al.*prove that the Chord system [45] is resilient against random faults in the fail-stop model, Hildrum and Kubiatowicz [16] proved the resilience of Pastry and Tapestry. It does not seem likely that these systems could be made resistant under false message injection without a significant change in their design. Fiat *et al.*[44, 11] propose a content addressable network that is robust against deletion and false message injection in the *worst case* scenario, i.e. when an adversary can choose which servers fail. In this model some constant fraction of the non-failed servers could be denied from accessing some of the data items. While their solution handles a more difficult model then ours, it has several disadvantages:

- It is not clear whether the system can preserve its qualities when servers join and leave dynamically.
- The linkage needed is $\Omega(\log^2 n)$.
- The construction is very complicated.

It is important to note that all constructions (including ours) assume that the construction itself was done properly; i.e. that during the Join/Leave operations nodes followed the protocol. Designing a protocol that is resilient against adversarial behavior in the Join operation seems to be a very difficult task.

6.2 The Overlapping Distance Halving DHT

Our construction (yet again) is a discretization of a continuous graph. The continuous graph we use is the *same* continuous graph used to build the DH DHT in Section 2.1. The difference is in the discretization technique.

The Discrete graph G Each server V_i $(1 \leq i \leq n)$ in the graph is associated with a *segment* $s(V_i) \stackrel{def}{=} [x_i, y_i]$. These segments should have the following properties:

- *Property I* - The set of points $\vec{x} = x_1, x_2, \ldots, x_n$ is evenly distributed along I. Specifically we desire that every interval of length $\frac{\log n}{n}$ contains $\Theta(\log n)$ points from $\vec{x}$. The point x_i is fixed and does not change as long as V_i is in the network.
- *Property II* - The point y_i is chosen such that the length of each segment is $\Theta(\frac{\log n}{n})$. It is important to notice that for $i \neq j$, $s(V_i)$ and $s(V_j)$ may *overlap*. The point y_i will be updated as servers join and leave the system. The precise manner in which y_i is chosen and updated is described in the next section.

The edge set of G is defined as follows. A pair of vertices (servers) V_i, V_j is an edge in G if $s(V_i)$ and $s(V_j)$ are connected in G_c or if $s(V_i)$ and $s(V_j)$ overlap. The edges of G are bi-directional. As before, a point $z \in I$ is said to be *covered* by V_i if $z \in s(V_i)$. The mapping of data items to servers is done as before, server V_i stores all data items $\mathcal{D}$ for which $h(\mathcal{D}) \in s(V_i)$ when h is some hash function. We observe the following:

1. Each point in I is covered by $\Theta(\log n)$ servers of G. This means that each data item is stored at $\Theta(\log n)$ servers.
2. Each server in G has degree $\Theta(\log n)$.

Join and Leave: Our goal in designing the Join and Leave operations is to make sure that properties I,II remain valid. When server V_i wishes to join the system it does the following:

Join Algorithm V_i

1. Choose at random $x_i \in [0, 1)$
2. Calculates a variable q_i which is an estimation of $\frac{\log n}{n}$.
3. Set $y_i = x_i + q_i \mod 1$.
4. Updates all the appropriate neighbors according to the definition of the construction.
5. The neighbors may decide to update their estimation of $\frac{\log n}{n}$ and therefore change their y value.

When server V_i wishes to leave the system (or is detected as down) all its neighbors should update their routing tables and check whether their estimation of $\frac{\log n}{n}$ should change. If so they should change their y value accordingly. The following lemma is straight forward:

Lemma 6.1. *If n points are chosen randomly, uniformly and independently from the interval $[0, 1]$ then with probability $1 - \frac{1}{n}$ each interval of length $\Theta(\frac{\log n}{n})$ contains $\Theta(\log n)$ points.*

If each server chooses its x-value uniformly at random from I then property-I holds. Observe that if each server's estimation of $\frac{\log n}{n}$ is accurate within a multiplicative factor then property II holds as well. The procedure for calculating q_i is very simple. Assume x_j is the predecessor of x_i along I. It is proven in [29] that with high probability

$$\log n - \log\log n - 1 \leq \log\left(\frac{1}{d(x_i, x_j)}\right) \leq 3\log n$$

Conclude that server V_i can estimate $\log n$ within a multiplicative factor simply by inverting the distance between its x-value and the x-value of its predecessor. Call this estimation α_i. A multiplicative estimation of $\log n$ implies a *polynomial* estimation of n, therefore an additional idea should be used. Let q_i be such that in the interval $[x_i, x_i + q_i]$ there are *exactly* α_i different x-values. A direct consequence of Lemma 6.1 is the following:

Lemma 6.2. *With high probability the number q_i estimates $\frac{logn}{n}$ within a multiplicative factor.*

When a server joins or leaves the system at most $O(\log n)$ servers need to update their q value. So with high probability property II holds as well.

Mapping the data items to servers The mapping of data items to servers is done in the same manner as previously. First data items are mapped into the interval I using a hash function. Server V_i should hold all data items mapped to points in $s(V_i)$.

All servers holding the same data item are connected to one another so they form a clique. Once a server storing a data item was located, the remaining servers storing the same data item are quickly located as well. This means that accessing different servers associated with the same data item in parallel can be simple and efficient. It suggests storing the data using an *erasure correcting code*, (for instance the digital fountains suggested by Byers *et al.*[5]) and thus avoid the need for replication. The data stored by any small subset of the servers suffices to reconstruct the data item. It is known that often the use of erasure correcting codes is more efficient than replication. For instance, Weatherspoon and Kubiatowicz [47] claim that an erasure correcting code may improve significantly the bandwidth and storage used by the system.

6.3 The Lookup Operation

The routing properties of the continuous graph G_c were discussed in Section 2.2. Assume processor V_i looks up point $y \in I$. Recall that Claim 2.4 implies that there is a point $z \in s(V_i)$ such that there is a path of length $O(\log n)$ between z and y in G_c. Call this path the *canonical path*. The canonical path exists is G_c, yet by the definition of G, if (a, b) is an edge in G_c, a is covered by V_i and b is covered by V_j then the edge (V_i, V_j) exists in G. This means that the canonical path can be *emulated* by G.

Simple Lookup Every point in I is covered by $\Theta(\log n)$ servers. This means that when server i wishes to pass a message to a server covering point $z \in I$ it has $\Theta(\log n)$ *different* neighbors that cover z. In the Simple Lookup it chooses *one* of these servers at random and sends the message to it. The following theorem follows directly from Claim 2.4 and Theorem 2.7.

Theorem 6.3. *Simple Lookup has the following properties:*

1. *The length of each lookup path is at most* $\log n + O(1)$. *The message complexity is* $\log n + O(1)$.
2. *If* V_i *is chosen at random from the set of servers and* y *is chosen at random from* I*, then the probability a given server participates in the lookup is* $\Theta(\frac{\log n}{n})$.

The following theorem describes the fault tolerance properties of the lookup:

Theorem 6.4. *If each server fail-stops independently with fixed probability* p*, then for sufficiently low* p *(which depends entirely on the parameters chosen when constructing* G*), with high probability each surviving server can locate every data-item.*

Note that the theorem holds even if failures are correlated (e.g. by a power failure). As long as the failures are independent of the random choices within the system, the randomization of the Join algorithms implies that I.D's of the failed servers is a random set.

Proof. We prove the following claim:

Claim 6.5. *If* p *is small enough, then w.h.p every point in* I *is covered by at least one server.*

Proof. Assume for convenience that $x_1 < x_2 < \cdots < x_n$. Each point in an interval $[x_i, x_{i+1}]$ is covered by the *same* set of $\Theta(\log n)$ servers. Call this set S_i. We have

$$\Pr[\text{ All servers in } S_i \text{ were deleted }] = p^{\Theta(\log n)}$$

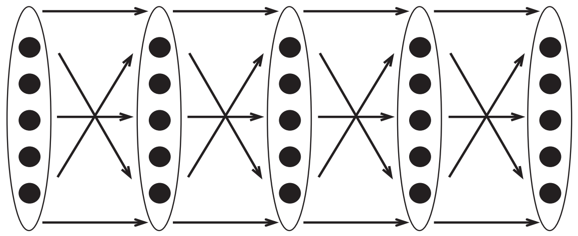

Figure 4: The message is sent through *all* the servers covering the path.

Therefore for sufficiently small p this probability is smaller than n^{-2}. Applying the union bound over all i yields that with probability greater than $1 - \frac{1}{n}$ every point in I is covered by at least one server. It is important to notice that for an arbitrary value of p it is possible to adjust the q values, so that each point in I is covered by sufficiently many servers, and the claim follows. $\square$

For every edge (a, b) in G_c there exists at least one edge in G whose servers cover a and b, therefore the canonical path could be emulated in G and the simple lookup succeeds. We stress that after the deletions the lookup still takes $\log n$ time and $\log n$ messages. Furthermore the average load induced on each server does not increase significantly. $\square$

False Message Resistant Lookup Now we assume that a failed server may generate arbitrarily false data items. We wish to show that every server can find all *correct* data items w.h.p. Just as in the simple lookup, the false message resistant lookup between the server V and $y \in I$ emulates the path between $s(V)$ and y in the continuous graph. The main difference is that now when server V_i wishes to pass a message to a server covering point a it will pass the message to *all* $\Theta(\log n)$ servers covering a. At each time step each server receives $\Theta(\log n)$ messages, one from each server covering the previous point of the path. The server sends on a message only if it were sent to it by a *majority* of servers in the previous step.

Theorem 6.6. *Assume each server fails with some small enough probability p. The false message resistant lookup has the following properties:*

1. *With high probability all surviving servers can obtain all correct data items.*
2. *The lookup takes (parallel) time of $\log n$.*
3. *The lookup requires $O(\log^3 n)$ messages in total.*

Proof. As before, Statement (2) follows directly from Claim 2.4. Statement (3) is correct since each edge of the canonical path in G_c translates to a bipartite complete graph with $O(\log n)$ nodes and $O(\log^2 n)$ edges, and a message is passed along each of these $O(\log^2 n)$ edges. It remains to prove Statement (1). Claim 6.5 in fact shows that if each server fails with probability p, then for sufficiently small p (which depends entirely on the parameters chosen when constructing G) it holds that with high probability every point in I is covered by a majority of non-failed servers. Now the proof of Theorem 6.6 is straight forward and is done by induction on the length of the path. Every point of the canonical is covered by a majority of good servers, therefore every server along the path receives a majority of the authentic message. It follows that with high probability *all* servers can find *all* true data items. $\square$

The easy proofs of Theorems 6.4 and 6.6 demonstrate the advantage of designing the algorithms in G_c and then migrating them to G. Proving the robustness of G_c is a straight forward argument.

7 Emulating General Graphs - Smoothness is Everything

In this section we show how our techniques can be used to dynamically construct an overlay network which embeds *any* family of fixed degree graphs. We show that if it is possible to maintain a smooth partitioning of $[0, 1)$ plus a lookup operation such as the DH construction offers, then it is possible to emulate any bounded degree graph with fixed cost. General techniques for constructing network topologies were independently suggested by Abraham *et al.*[1].

Let $\{G_1, G_2, \ldots\}$ be an infinite family of graphs with maximum degree d. Assume that G_i has 2^i vertices, denoted $u_1, u_2, \ldots, u_{2^i}$. We first describe a static simulation and then show how to make it dynamic. Let $\vec{x}$ be a smooth set of n points in $[0, 1)$ and associate a node (server) V_j with each point. We define a graph $G_{\vec{x}}$ on $V = \{V_1, \ldots V_n\}$ that emulates $G_{\lceil \log n \rceil}$. For each k, define the function Φ_k from the 2^k nodes of G_k to the n servers of $G_{\vec{x}}$ as follows:

$$\Phi_k(u_j) = V_i \quad \text{if } \tfrac{j}{2^k} \in s(x_i).$$

If $\vec{x}$ is smooth, then the function Φ_k spreads the nodes of G_k evenly among the servers. Note that the function Φ_k can be computed locally, i.e. each server V_i can determine based on x_i and x_{i+1} which nodes in G_k are mapped to it. The edges of $G_{\vec{x}}$ are defined as:

$$E(G_{\vec{x}}) = \{(V_i, V_j) \mid \exists (u_\ell, u_m) \in E(G_k)\ ,\ \Phi_k(u_\ell) = V_i, \Phi_k(u_m) = V_j\}\,.$$

If $(u_\ell, u_m) \in E(G_k)$ and $\Phi_k(u_\ell) = V_i$ and $\Phi_k(u_m) = V_j$ then we say that edge (u_ℓ, u_m) is *simulated* by (V_i, V_j). Let $k = \lceil \log n \rceil$ and let ρ be the smoothness of $\vec{X}$. It is straightforward to verify the following properties:

1. Every server in $G_{\vec{x}}$ simulates at most $\rho + 1$ servers in G_k: if a nodes are mapped to sever i then $\frac{a-1}{2^k} \leq x_{i+1} - x_i$ and by smoothness $x_{i+1} - x_i \leq \rho/n \leq \rho/2^k$.
2. Every edge in $G_{\vec{x}}$ simulates at most ρ^2 edges in G_k: follows from (1).
3. The degree of $G_{\vec{x}}$ is at most $\rho \cdot d$: follows from (1).

In other words if $\vec{x}$ is smooth then $G_{\vec{x}}$ is a *real time emulation* of G_k. In Particular this means that any computation performed by G_k, could be performed by $G_{\vec{x}}$ in constant slow down. (See [28] and [23] for an overview on the literature of real time emulations.)

Suppose that the servers have a mechanism to dynamically maintain a smooth partition $\vec{x}$ as servers join and leave as well as a lookup mechanism, i.e. each server can find the server in charge of point $x \in [0, 1)$ (as in the DH construction). Then, assuming all servers know what n is, each server can calculate separately which are its neighbors in $G_{\vec{x}}$ using the lookup service. Furthermore, they can maintain $G_{\vec{x}}$ as changes occur and the cost is not more than $O(\rho)$ per change.

Next we remove the assumption that all servers know the value of n, using the smoothness (there are other methods for doing so, as discussed in [29] and [17]). A smooth $\vec{x}$ implies that each server V_i can estimate the value of n setting $n_i = \frac{1}{|s(V_i)|}$ (n_i is server S_i's guess to n). By definition of smoothness $\max_{i,j} \frac{n_i}{n_j} = \rho(\vec{x})$. Therefore it holds that for all $1 \leq i \leq n$ we have $\log n_i - \log \rho \leq \log n \leq \log n_i + \log \rho$. If $\vec{x}$ is guaranteed to have smoothness of at most ρ, then each server can calculate a list of length $2 \log \rho$ that contains $\lceil \log n \rceil$. Each server now sets edges according to every index in its list; i.e. the edges each server would open would result from the union of the Φ's on its list.

Theorem 7.1. *When $G_{\vec{x}}$ is constructed and maintained as described above it holds that*

1. *The degree of $G_{\vec{x}}$ is at most $2d \cdot \rho \log \rho$.*

2. *If ρ is a constunt then the graph $G_{\vec{x}}$ can emulate in real time the graph $G_{\lceil \log n \rceil}$.*

Theoretically speaking, this result implies that considering scalable systems separately is superfluous; i.e. for any problem it is possible to come up with a solution in a static environment and then make it dynamic via this technique. The main disadvantage of this technique is that the dependency on the smoothness is heavier, so tailored designs are indeed interesting.

Acknowledgments We gratefully thank Dan Boneh, Frank Dubek, Cynthia Dwork, Andrew Goldberg, Jason Hartline, Anna Karlin, Dalia Malkhi, Frank McSherry and Kfir Zigdon for their valuable help. We thank the diligent referees for their useful comments.

References

[1] Ittai Abraham, Baruch Awerbuch, Yossi Azar, Yair Bartal, Dahlia Malkhi, and Elan Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 40, 2003.

[2] William Aiello, Baruch Awerbuch, Bruce M. Maggs, and Satish Rao. Approximate load balancing on dynamic and asynchronous networks. In *ACM Symposium on Theory of Computing*, pages 632–641, 1993.

[3] Noga Alon and Joel H. Spencer. *The Probabilistic Method.* John Wiley & Sons, second edition, 2000.

[4] James Aspnes and Udi Wieder. The expansion and mixing time of skip graphs with applications. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 126–134, 2005.

[5] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[6] Jin-yi Cai. Essentially every unimodular matrix defines an expander. *Theory of Computing Systems*, 36(2):105–135, 2003.

[7] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX 1996 annual technical conference*, pages 153–163, 1996.

[8] Edith Cohen and Haim Kaplan. Balanced-replication algorithms for distribution trees. In *European Symposium on Algorithms (ESA)*, pages 297–309, September 2002.

[9] Krzysztof Diks and Andrzej Pelc. Optimal adaptive broadcasting with a bounded fraction of faulty nodes. *Algorithmica*, 28(1):36–50, 2000.

[10] Devdatt P. Dubhashi and Alessandro Panconesi. Concentration of measure for the analysis of randomised algorithms. Book Draft, October 2005.

[11] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Symposium on Discrete Algorithms (SODA)*, pages 94–103, 2002.

[12] Pierre Fraigniaud and Philippe Gauron. an overview of the content-addressable network d2b. In *Symposium on Principles of Distributed Computing (PODC)*, page 151, 2003.

[13] Joel Friedman. A proof of Alon's second eigenvalue conjecture. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing (STOC)*, pages 720–724, 2003.

[14] Ofer Gabber and Zvi Galil. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Science*, 22:407–420, 1981.

[15] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, 2004.

[16] Kirsten Hildrum and John Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *DISC*, pages 321–336, 2003.

[17] Keren Horowitz and Dahlia Malkhi. Estimating network size from local information. *Information Processing Letters*, 88(5):237–243, 2003.

[18] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Second International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 98–107, 2003.

[19] David Karger, Eric Lehman, F. Thomson Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[20] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 36–43, 2004.

[21] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *Seventeenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, 2005.

[22] Jon Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, pages 163–170, 2000.

[23] Richard R. Koch, F. Thomson Leighton, Bruce M. Maggs, Satish Rao, Arnold L. Rosenberg, and Eric J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44(1):104–147, January 1997.

[24] Michael Larsen. Navigating the cayley graph of $\mathrm{SL}_2(\mathbb{F}_p)$. *International Mathematics Research Notices*, 27:1465–1471, 2003.

[25] Ching Law and Kai-Yeung Siu. Distributed construction of random expander graphs. In *Proceedings of IEEE INFOCOM*, 2003.

[26] F. Thomson Leighton. *Introduction to parallel algorithms and architectures : arrays, trees, hypercubes*, chapter 3.3. Morgan Kaufmann, San Mateo, CA, 1992.

[27] F. Thomson Leighton and Bruce Maggs. Expanders might be practical: Fast algorithms for routing around faults in multibutterflies. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 384–389, 1989.

[28] Bruce M. Maggs and Eric J. Schwabe. Real-time emulations of bounded-degree networks. *Information Processing Letters*, 66(5):269–276, 1998.

[29] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC*, pages 183–192, 2002.

[30] Gurmeet S. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 197–205, 2004.

[31] G. A. Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, (9(4)), October - December 1973.

[32] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 53–65, 2002.

[33] Michael Mitzenmacher, Andrea Richa, and Ramesh Sitaraman. The power of two random choices: A survey of the techniques and results. In *Handbook of Randomized Computing, P. Pardalos, S. Rajasekaran, and J. Rolim, Eds. Kluwer.*, 2000.

[34] Uri Nadav and Moni Naor. Fault-tolerant storage in a dynamic environment. In *DISC*, pages 390–404, 2004.

[35] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 50–59, 2003.

[36] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Second International Workshop on Peer-to-Peer Systems*, pages 88–97, February 2003.

[37] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing (prelimenary version appeared in PODC 2003*, 17(4):311–322, 2005.

[38] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Song N. Chiu. *Spatial Tessellations — Concepts and Applications of Voronoi Diagrams*. Wiley, Chichester, second edition, 2000.

[39] C. Greg Plaxton and Rajmohan Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *IEEE Symposium on Foundations of Computer Science*, pages 570–579, 1996.

[40] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[41] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proc ACM SIGCOMM*, pages 161–172, San Diego CA, Augost 2001.

[42] Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. Routing algorithms for dhts: Some open questions. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 45–52, 2002.

[43] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[44] Jared Saia, Amos Fiat, Steven Gribble, Anna R. Karlin, and Stefan Saroiu. Dynamically fault-tolerant content addressable networks. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 270–279, 2002.

[45] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[46] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[47] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 328–338, 2002.

[48] Ben Y. Zhao and John Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB CSD 01-1141, University of California at Berkeley, Computer Science Department, 2001.