

Novel Architectures for P2P Applications: the Continuous-Discrete Approach

Moni Naor*

Udi Wieder *

Abstract

We propose a new approach for constructing P2P networks based on a dynamic decomposition of a continuous space into cells corresponding to processors. We demonstrate the power of these design rules by suggesting two new architectures, one for DHT (Distributed Hash Table) and the other for dynamic expander networks. The DHT network, which we call **Distance Halving**, allows logarithmic routing and load, while preserving constant degrees. 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. We show it is possible to reduce the dilation and the load of the DHT with a small increase of the degree. We present a provably good protocol for relieving hot spots and a construction with high fault tolerance. Finally we show that, using our approach, it is possible to construct any family of constant degree graphs in a dynamic environment, though with worst parameters. Therefore we expect that more distributed data structures could be designed and implemented in a dynamic environment.

*The Weizmann Institute of Science, Rehovot 76100 Israel. {naor,uwieder}@wisdom.weizmann.ac.il

1 Introduction

Consider an overlay network in which processors are allowed to join and leave the system at any time. The problem we deal with is how to maintain a network structure and functionality in such a dynamic environment where the participants of the network change over time. A common notion that sometimes is used to describe these networks is Peer-to-peer (P2P) systems. Peer-to-peer networks are characterized by the lack of central control or a-priori hierarchical organization. Moreover a P2P system is usually expected to scale gracefully as the size of the network grows. Scalable networks have attracted a considerable amount of attention, partly due to the wide popularity of internet based file sharing protocols.

Recently a new and powerful generation of scalable overlay networks were proposed, that support a *distributed hash table* (DHT) functionality. Among them are Tapestry [24], Chord [21], Pastry [19], CAN [18] and Viceroy [14]. In these systems data items are associated with a key and each processor in the system is responsible for storing a certain range of keys. These systems support a routing protocol that allows users to find a machine responsible for a required key. The methodology used in designing these graphs can be roughly described as follows: first find a static *family* of graphs in which there are good protocols for performing the desired tasks, then show how to construct in distributed manner, a network whose topology approximates the topology of these graphs. CAN approximates the d -dimensional torus. Chord and Pastry approximate the hypercube and Viceroy approximates the butterfly. The parameters in which a DHT is measured by include the following metrics:

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. We define this notion formally in Section 2.2.

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 linkage; i.e. the degree of the graph that represents the network, should be low.

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. This is called the *dilation* of the network.

Fault tolerance: The service should function well after some of its servers/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 keeping small the number of data items each server has to store.

Viceroy [14] presented for the first time a system that maintains with high probability a logarithmic diameter, a constant degree and given these parameters an optimal (up to constants) congestion. Table 1 summarizes the performance of different constructions under these parameters.

1.1 Our Contributions

Our contribution is twofold. First we provide a framework and a set of design rules for constructing scalable networks. We rely on constructions over a continuous space in order to build discrete protocols. Previous works (such as Chord [21], Viceroy, [14], CAN [18], and consistent hashing [8]) have used continuous analogues to construct discrete structures as well. We differ by designing and proving the algorithms (e.g.

<i>Lookup Scheme</i>	<i>dilation</i>	<i>congestion</i>	<i>linkage</i>
Chord [21]	$\log n$	$(\log n)/n$	$\log n$
Tapestry [24]	$\log n$	$(\log n)/n$	$\log n$
CAN [18]	$dn^{1/d}$	$dn^{1/d-1}$	d
Small Worlds [9]	$\log 2n$	$(\log 2n)/n$	$O(1)$
Viceroy [14]	$\log n$	$(\log n)/n$	$O(1)$
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.

for routing and caching) themselves in the continuous structure, and then emulate them in the discrete one. These design rules prove to be very simple and powerful. Secondly we demonstrate how these design rules can be implemented by providing scalable networks with excellent parameters. We describe a DHT which we call **Distance Halving (DH)**. The DH DHT is very simple, yet it has logarithmic diameter and congestion and a small constant linkage. Furthermore, we also show how dynamic caching with provable performance against worst case loads can be implemented in such a system. We then show how a slight modification of the network results in network which is resilient against the (possibly Byzantine) failure of random subsets of servers. We demonstrate the flexibility provided by our design rules by building a network guaranteed to be an expander. Finally we show how *any* family of bounded degree graphs can be emulated.

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. Assume some fixed continuous graph G_c for which the vertex set is I . Each point 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 processor 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. If the cells which compose I are allowed to *overlap* then the resulting graph would be fault tolerant.

Sections 2 and 3 describe the DH DHT and its various algorithms assuming smoothness. Section 4 describes how smoothness is achieved. Section 5 generalizes the approach to higher dimensions. Section 6 describes a fault tolerant construction, and Section 7 shows how to approximate *any* family of graph.

In light of this framework...

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 the out-degree of each point is 2 while the in-degree is 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 . Now we show how to construct a *discrete* Distance Halving graph. Denote by \vec{x} a set of n points in I . The point x_i may be the i.d of processor v_i or some hash of its i.d. The points of \vec{x} divide I into n segments. Define the *segment* of x_i to be $s(x_i) = [x_i, x_{i+1})$ ($i = 1 \dots n - 1$) and $s(x_n) = [x_{n-1}, 1) \cup [0, x_1)$.

In the *discrete* Distance Halving graph $G_{\vec{x}}$, each processor v_i is associated with the segment $s(x_i)$. 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, v_1) are added such that $G_{\vec{x}}$ contains a ring. The ring edges are anti-parallel directed edges. If processor v_i wishes to enter the system it does the following:

Algorithm Join

1. Choose x_i and set $v_i.id \leftarrow x_i$. (How to choose x_i is not shown here. Various options for doing that are discussed in Section 4).
2. Let v_j be the processor such that $x_i \in s(x_j)$. Lookup v_j and receive from it the data items that are mapped to $s(x_i)$ and the addresses of all the neighbors v_i should have.
3. Inform the neighbors of v_i 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 processor v_i leaves the network, some existing processor should take over its segment. The simplest solution would be that the processor that is the predecessor of v_i on the ring enlarges its segment such that it includes $s(v_i)$. More sophisticated solutions are discussed in Section 4.

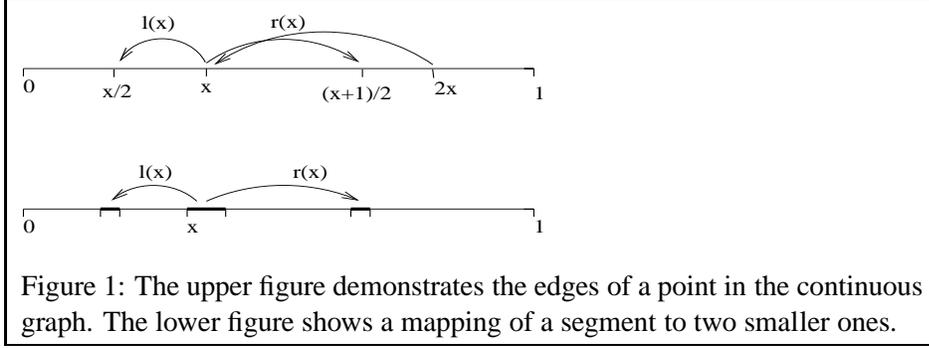
Theorem 2.1. *The total number of edges in $G_{\vec{x}}$ without the ring edges is at most $3n - 1$.*

Proof. The proof is by induction on n . If $n = 1$ then there are two self edges. Assume \vec{x} has $n - 1$ points, and point n is now added. The degree of the continuous graph is 3. The segment that the new point holds was previously held by another point, therefore the addition of point n can add at most 3 new edges to the graph. \square

The *average* degree of the graph is at most 6. In order to bound the *maximum* degree, 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 $\rho(\vec{x})$ is a 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 maximal out-degree of $G_{\vec{x}}$ without the ring edges is at most $\rho(\vec{x}) + 2$, the maximal in-degree without the ring edges is at most $2\rho(\vec{x}) + 1$.*

Proof. Let i be such that $|s(x_i)|$ is maximal. The length of the minimal 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 $\frac{1}{2}\rho + 1$ different segments that intersect the interval $r(s(x_i))$. The same argument applies for $\ell(s(x_i))$. Adding them up results with the bound on the maximal out-degree.

Similar argument yields that the in-degree is bounded by $2\rho(\vec{x}) + 1$. \square

Mapping the data items to processors: The mapping of data items to nodes is done in the same manner as other constructions of distributed hash tables (such as consistent hashing [8], Chord [21], Viceroy [14] and CAN [18]). First data items are mapped into the interval I using a hash function. Processor v_i should hold all data items mapped to points in $s(v_i)$. We assume that h is some k -wise independent function which is chosen at the construction of the system, and is given to every processor upon joining.

The De-Bruijn Graph The Distance Halving construction generalizes the known De-Bruijn graph. The r -dimensional De-Bruijn graph consists of 2^r processors and 2^{r+1} directed edges. Each node corresponding to an r -bit binary string. There is a directed edge from each node $u_1u_2 \cdots u_r$ to nodes $u_2 \cdots u_{r-1}0$ and $u_2 \cdots u_{r-1}1$. See Leighton [11] for an overview of various properties of this graph. 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 $G_{\vec{x}}$ is isomorphic to the r -dimensional De-Bruijn graph.

2.2 The Lookup Operation

We set some notations that would be useful in the future. Define $d(x_i, y)$ to be $|x_i - y|$. Let σ denote a sequence of binary digits, and σ_t denote its prefix of length t . For every point $y \in I$ we define $\sigma_t(y)$ in the following manner: $\sigma_0(y) = y$, $(\sigma_t.0)(y) = \ell(\sigma_t(y))$, $(\sigma_t.1)(y) = r(\sigma_t(y))$. In other words $\sigma_t(y)$ is the point reached by a walk that starts at y and proceeds according to σ_t when 0 represents ℓ and 1 represents r .

Routing properties of the continuous DH graph: The following claim justifies the name ‘Distance Halving’:

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

$$d(r(y), r(z)) = d(\ell(y), \ell(z)) = \frac{1}{2}d(y, z) \tag{1}$$

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

The binary representation of y represents the direction of the single in-degree edge entering y , i.e. when walking *backwards* from y along the in-degree edge, the direction of the i^{th} edge (left or right) is determined by the i^{th} bit of the binary representation of y . The following claim is used to find short paths between different segments of the continuous graph.

Claim 2.4. *Let $y, z \in I$. Let σ be the binary representation of y , then for all t it holds that $d(y, \sigma_t(z)) \leq 2^{-t}$*

Proof. Let y' be such that $\sigma_t(y') = y$; i.e. a walk that starts at y' and follows the binary representation of the prefix of length t of y , reaches y . We have: $d(\sigma_t(z), y) = d(\sigma_t(z), \sigma_t(y'))$. By Claim 2.3 it holds that $d(\sigma_t(z), \sigma_t(y')) = 2^{-t}d(z, y') \leq 2^{-t}$. \square

The previous two claims show 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 procedures described in Claims 2.3 and 2.4. Assume processor v_i wishes to lookup the point y . The point y can either be an i.d number of a processor or a key of a file (it may be that $y = h(\text{file-name})$ where h is a hash function) and therefore can be an arbitrary point on the line, in which case v_i seeks the processor 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.

Greedy Lookup: Claim 2.4 states that from any point in I it is possible to reach a point that is close to y by walking according to the binary representation of y . If $y \in s(x_i)$ for some processor, then $s(x_i)$ would also contain points close to y . This observation is used to emulate the continuous lookup in $G_{\bar{x}}$. In the protocol the message is routed via the *backward* edges, therefore we assume that every edge is antiparallel. The header of the message should hold the destination y and the current position of the message on I .

Greedy Lookup (x_i, y)

1. Let w be the point in the middle of $s(x_i)$. Let σ be the binary representation of w . Calculate the minimum t such that $\sigma_t(y)$ is in $s(x_i)$.
2. Let $z = \sigma_t(y)$. It must be that $z \in s(x_i)$. Start moving from z (i.e. from $s(x_i)$ and therefore from v_i) on the *backward* edges. Each time the header of the message should be updated so it holds the current position of the message on I . After t steps the processor holding y is reached .

The length of the lookup path is determined by t . If the segments are smooth, then $|s(x_i)|$ can not be too small thus t must be small as well. A direct corollary of Claim 2.4 is that $t = -\log(|s(x_i)|) + 1$ 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 $\log n + \log \rho + 1$.*

Next we analyze the congestion of Greedy Lookup.

Definition 2. The *congestion* of vertex v_i is the probability v_i is active in a routing between two randomly chosen processors. The congestion of the network is the maximum congestion over all its processors.

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

Lemma 2.6. *Let y, z be chosen randomly from I , and ρ be constant. The probability that a processor participates in a lookup of length $\log n + \log \rho$ that starts at y and approaches z is at most $\Theta(\frac{\log n}{n})$.*

Proof. Fix a processor v . The random choice of z corresponds to the choice of the random string σ of length t (where t is at most $\log n + \log \rho$). In order for v to participate in the i^{th} step of the lookup two events must occur:

1. There is an interval of length $2^i |s(v)|$ from which it is possible to reach $s(v)$ in i steps. It is necessary that y falls *within* that interval.
2. Once y is chosen, it is necessary that the i first bits of the random string σ would be such that a message starting from y would actually reach $s(v)$.

Both events are independent from one another, therefore for each $i = 0 \dots t$ the probability of processor v to participate in lookup in the i^{th} step is $|s(v)| \cdot 2^i \cdot 2^{-i} = |s(v)|$. When ρ is constant $|s(v)| = \Theta(\frac{1}{n})$. Adding the probabilities for all i yields the result. \square

Theorem 2.7. *Let \vec{x} be a smooth set of points. The congestion of $G_{\vec{x}}$ under Greedy Lookup is $\Theta(\frac{\log n}{n})$.*

In view of the previous lemma, Theorem 2.7 is straightforward. The same line of arguments work. The detailed proof would appear in the full version of the paper. Note that there is no uncertainty in the result of Theorem 2.7. If \vec{x} is smooth then the congestion is low for *all* processors with certainty.

Distance Halving Lookup: The Distance Halving lookup scheme enjoys small congestion even in a worst case. 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. The header of a message should contain the source x_i , the target y and a random string σ . Upon receiving a message a processor does the following:

Distance Halving Lookup - Phase I

1. Check if $\sigma_t(y)$ is covered by the current segment or by one of the neighboring segments. If so move the message to the processor responsible of $\sigma_t(y)$ and move to phase II.
2. Randomly choose a bit and add it to σ the random string in the header of the message.
3. Calculate the new $\sigma_t(x_i)$ and update the header accordingly.
4. Send the message to the neighbor covering the point $\sigma_t(x_i)$. (An edge must exist).

In the second phase the message moves *backwards* from $\sigma_t(y)$ to y . Each step the processor handling the message deletes the last bit in σ . It is convenient to think of it as two messages moving simultaneously, one from the source and one from the target. Both of them move according to the same sequence σ . According to Claim 2.3 each step the distance between them reduces by half, until they meet at a random point in the middle. The following theorem is a direct result 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. Fix a processor v . Let L_i be the random variable indicating whether the lookup reached v in step i . We claim that $\Pr[L_i] = \Theta(|s(v)|)$. This is proved by induction on i . For $i = 1$ it is immediate. For a message to reach v on step i it must be on an interval of size $2|s(v)|$ on step $i - 1$ and then go to the direction of $s(v)$. Therefore by the i.h we have $\Pr[L_i] = \frac{1}{2} \cdot \Theta(2|s(v)|) = \Theta(|s(v)|)$. Summing up over i proves that the probability v participates in the first phase of the routing is $\Theta(\frac{\log n}{n})$. The analysis of the second phase is similar. \square

The Distance Halving Lookup is similar to the routing scheme suggested by Valiant [22] for the hypercube. Therefore it is not surprising that it imposes small congestion for worst case ‘permutation’ routing. Assume for all i processor v_i initiates a lookup for data item m_i ; i.e. processor v_i seeks for the processor holding the point $h(m_i)$ where h is a hash function. Assume that all data items are different, that is $m_i \neq m_j$ for all $i \neq j$.

Theorem 2.10. *If \vec{x} is smooth, and h is $\log n$ -wise independent, then with high probability each processor served $O(\log n)$ messages.*

Proof. Fix a processor v . In the following we prove that in the first phase, the expected number of lookups that v participates in is $O(\log n)$. Let $L_i^{s(v)}$ be the random variable stating the number of lookups that reach $s(v)$ at the i^{th} step. We claim that $E[L_i^{s(v)}] \leq \alpha n |s(v)| = O(1)$ for some constant α . By definition $L_0^{s(v)} = 1$. For a message to reach $s(v)$ in the i^{th} step, it must be that on step $i - 1$ the message was in an interval of length $2|s(v)|$ and then had to move to $s(v)$. Call this interval Q .

$$\begin{aligned} E[L_i^{s(v)}] &= \frac{1}{2} L_{i-1}^Q \\ E[L_i^{s(v)}] &= \frac{1}{2} E[L_{i-1}^Q] \end{aligned}$$

The induction hypothesis states that

$$\frac{1}{2} E(L_{i-1}^Q) \leq \frac{1}{2} \alpha n |Q| = \alpha n |s(v)| = O(1)$$

Summing over i yields that the expected number of lookups that pass through v in the first phase is $O(\log n)$. The analysis of the second phase is similar.

Next we prove that w.h.p the actual number of lookups that pass through v is indeed $O(\log n)$. Let p_i ($i = 1 \dots n$) indicate the probability that message i passes through $s(v)$ during the first phase of the lookup. By the previous claim we know that

$$\sum_{i=1}^n p_i = O(\log n)$$

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 would allow us to use the union bound over all processors thus proving the bound for the first phase.

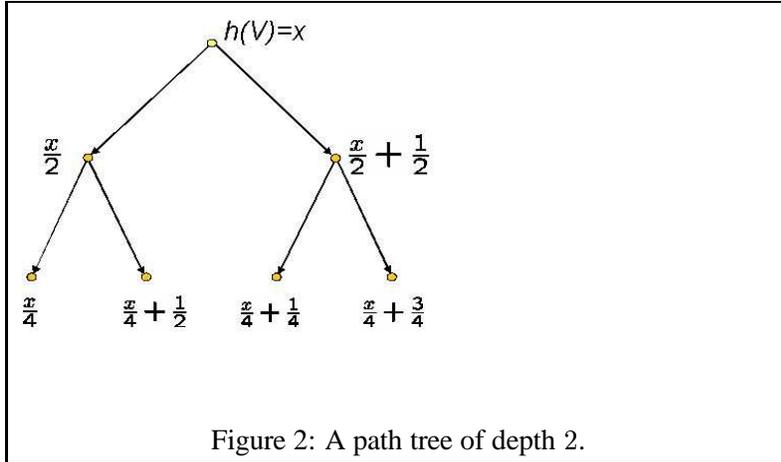
In the second phase it is not true that the paths messages take are independent, but rather are $\log n$ -wise independent. Using a $\log n$ moment inequality we have

$$\Pr \left[\left| \sum p_i - E[\sum p_i] \right| \geq c \log n \right] \leq \frac{\Theta(\log n)^{\log n}}{(c \log n)^{\log n}}$$

Once again if c is chosen to be large enough, then the union bound would imply that w.h.p *all* processors handled $O(\log n)$ messages. \square

Theorem 2.10 demonstrates that in some sense the Distance Halving Lookup is good in a worst case. 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 *not* an oblivious routing; i.e. the edge in which the message is sent, is not a function of the destination only but is also a function of the numeric value of $x_i(\sigma_i)$.
- 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 routing algorithms require a writing operation for every packet.



2.3 Reduced Lookup Path Length

We show how increasing the degree reduces the lookup length and the congestion. For any $c \geq 2$ construct a continuous graph with the following edges: $f_i(y) = \frac{y}{c} + \frac{i}{c}$ ($i = 0, 1, \dots, c-1$). Now Claim 2.3 translates to be $d(f_i(y), f_i(z)) = \frac{1}{c}d(y, z)$ and Claim 2.4 translates to be $d(y, \sigma_i(z)) \leq c^{-t}$. Therefore:

Theorem 2.11. *A discretization of the continuous graph would result with a graph of degree $\Theta(c)$ and with dilation $\log_c n$.*

Note that the diameter of any graph with degree c is at least $\log_c n$. Two interesting options are setting $c = \log n$ or $c = n^\epsilon$ (for some constant ϵ), as the first results with a lookup length of $\frac{\log n}{\log \log n}$, and the second with a lookup length of $O(1)$. It is worth noting that the analysis of the previous section shows that for each choice of c , if the points are smooth the congestion would be a $\Theta(\frac{\log_c n}{n})$.

3 Dynamic Caching - Relieving Hot Spots

A highly popular data item may cause a bottleneck at and around its location. This phenomena cannot be dealt with locally. It is necessary to replicate the popular data item to other processors, such that the load of handling all requests is distributed between a large number of processors. Various replication techniques were suggested in the literature ([8],[4],[3]), most notably Karger *et al* [8] suggest a caching capable of relieving hot spots in a large dynamic network. Their protocol makes a distinction between cache servers and users while ours makes no such distinction. In addition their protocol is used on top of the routing protocol while ours utilizes the properties of the Distance Halving routing protocol to perform the caching efficiently.

3.1 The Protocol

As usual we first describe the protocol in the continuous graph. Assume the popular data item is V and it is hashed to the value $y \in I$, so the processor which holds y also holds a copy of V .

Definition 3. The *path tree* rooted at y is the tree created by letting y be the root, and each node z in the tree is the father of $\ell(z), r(z)$. A node which holds the data item V is called an *active* node. The tree which consists of all the active nodes is the *active tree*.

A path tree of depth two is drawn in Figure 2. Observe that if the Distance Halving routing algorithm is used, then every request for V would reach y via a *random path* in the path tree. This observation suggests that it is wise to replicate V from the node of the tree down. Assume all processors of the system count¹ the same time epoch of length m . Assume that during this time epoch V is requested q times.

Continuous Hot Spots Protocol:

1. If $z \in I$ is a leaf in the active tree, and V was requested from z more than c times in a single epoch (c is some threshold), then V is replicated to $\ell(z), r(z)$ which become leaves of the active tree.
2. If z is the father of two leaves of the active tree, and during an epoch V was requested less than c times by both its children, then both children may delete V and cease to be active. 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 the discrete protocol (as usual) processor v_i emulates all the points in $s(v_i)$. The distance between any two points in the j^{th} layer of a path tree is at least 2^{-j} . It follows that if $j \leq \log n - \log \rho$, all the points in the j^{th} layer are covered by different processors of the discrete graph. It may be that a processor would cover one point from each layer of the tree. Consider for instance the processor which covers the point 0. If it covers the root of the tree it also covers all the points in the left branch. The following theorem bounds the congestion a single hot spot may cause.

Theorem 3.1. *W.h.p the number of times a processor handles a request for V in a time epoch is at most $c(\log q - \log c + O(1))$*

Proof. The proof follows directly from the following lemma in which we bound the growth of a branch in a single epoch. We assume that $c > \log q$.

Lemma 3.2. *If at the beginning of the epoch the lowest active processor is at layer i of the virtual tree, at the end of the epoch, with probability $1 - \frac{1}{q^c}$ the layer of the most distant active processor is at most $i + \log q - \log c + O(1)$.*

Clearly the worst case is when $i = 0$. Consider layer $\log q - \log c$. At this layer there are exactly $\frac{q}{c}$ processors. Since $c > \log q$ each processor received $\Theta(c)$ requests, therefore the longest branch stemming from such a processor is of length $O(1)$. □

Assume the threshold is set $c = \log n$, the demand is $q = n$ and that the hash function h which maps the data items into I is truly random. The following is the main theorem of this section, it proves that even on the worst case, w.h.p bottlenecks would not occur.

Theorem 3.3. *W.h.p the maximum number of messages a processor handles is at most $\Theta(\log^2 n)$.*

Proof Sketch: [Theorem 3.3] Assume there are requests for l data items. Data item i is requested q_i times so $\sum_{i=1}^l q_i = n$. Fix some processor v . First we bound the number of times v handles requests by supplying the requested data item; i.e. when v is a leaf of an active tree. Denote by p_i the number of points in the active tree of data item i that belong to $s(v)$. The processor v handles $c \cdot p_i$ requests out of the q_i requests of data item i . We assume that $q_i \geq \log n$, as otherwise data item i does not cause a hot spot.

Lemma 3.4. *The following two conditions hold:*

1. $0 \leq p_i \leq \log(\frac{q_i}{c}) + O(1)$
2. $E(p_i) \leq \Theta(\frac{q_i}{n \cdot c})$

¹We don't assume that the system is synchronized, this assumption is for convenience and does not play a major role.

Proof. The first assertion follows directly from Lemma 3.7. To prove the second assertion note that the probability that $s(v)$ contains a point from the j^{th} layer of the tree is $\Theta(\frac{2^j}{n})$. Summing this up for $(1 \leq j \leq \log(\frac{q_i}{c}))$ proves the lemma. \square

Lemma 3.5. *W.h.p $s(v)$ covers active points from at most $O(\log n)$ different active trees.*

Proof. Let b_i denote the random variable indicating whether v supplied the data item i . According to the previous lemma $\Pr[b_i = 1] \leq \frac{q_i}{nc}$. We also have $E[\sum b_i] \leq E[\sum b_i] \leq \frac{1}{c}$. If $c = \log n$ then a standard use of the Chernoff bound yields

$$\Pr \left[\sum b_i \geq \Theta(\log n) \right] \leq \frac{1}{n^2}$$

\square

Lemma 3.6. *Assume an active node data item i 's active tree is covered by v . The number of active nodes that are covered by v is a random variable dominated by a geometric distribution.*

Proof Sketch: This is the key lemma and its full proof is quite involved. We claim the following. The probability that v covers a point in layers $0, 1, \dots, j$ of an active tree, *conditioned on it covering a point in layer $j+1$* , is bounded by a constant independent of n . This immediately implies the lemma as it means that when going from the leaves up, in each layer there is a constant probability of never returning to $s(v)$. \square

Standard techniques of large deviations yield that with high probability the sum of $\Theta(\log n)$ geometrical random variables is at most $\Theta(\log n)$. We conclude that w.h.p v covers at most $\log n$ active nodes, therefore supplied data items at most $c \log n = \log^2 n$ times.

Next we bound the number of times a message passes through a processor as part of the routing protocol. This is done the same manner as the previous part. Divide the messages into bundles of size $\log n$. Each bundle received the data item from a *different* processor; i.e. if the same data item was returned by two different processors (both of them hold the data item) then we think of it as two different data item. Now we face the situation of Theorem 2.10. This means that with high probability each processor served at most $O(\log n)$ bundles, i.e. $O(\log^2 n)$ messages in total. \square

Now we bound the size of an active tree.

Lemma 3.7. *At the end of the epoch, with probability at least $1 - \frac{1}{n}$, the lowest active processor would be at layer $l = \frac{2 \log n}{c} + \log q - \log c$.*

Lemma 3.7. Let $r = 2^l$. There are r vertices in layer l . Let X denote the random variable indicating the number of requests a processor in layer l receives. The expectation of X is $\frac{q}{r}$. We have:

$$\Pr[X \geq c] \leq \Pr \left[X \geq \left(1 + \frac{cr}{q}\right) \frac{q}{r} \right].$$

Define $\epsilon = \frac{cr}{q}$. We have by Chernoff's bound:

$$\Pr \left[X \geq (1 + \epsilon) \frac{q}{r} \right] \leq e^{-\epsilon \log \frac{q}{r}} \leq \frac{1}{n^2}.$$

Applying the union bound yields that w.h.p all processors of layer l received less than c requests in the epoch, therefore by the end of the epoch they will not hold the value and would be inactive. \square

Lemma 3.7 implies that when $c = \log n$ the data item is replicated $\Theta(\frac{q}{c})$ times. Note that if servers are not allowed to serve more than c requests then $\Theta(\frac{q}{c})$ replications are necessary.

It is common that the *content* of the hot spot is dynamic as well. Consider for instance an internet site that describes a live basketball game. It is important to be able to *update* the popular data item quickly. The tree like structure of the cache means that the popular data item may be changed or altered efficiently. An update of the data item would take $O(\log \frac{q}{c})$ time and $O(\frac{q}{c})$ messages.

3.2 The Hypercube

In order to show the versatility of our design rules, we show how existing constructions could be described within our framework. Let I be the interval $[0,1)$ and y a point in I . For every $i \in \mathbb{N}$ let $f_i(y)$ be the point reached by changing the i^{th} bit of the binary representation of y ; i.e. $f_i(y) = y \oplus 2^{-i}$. The continuous graph G has I as its vertex set, and the functions f_i define its edges. It is easy to see that G is a continuous generalization of an infinite hypercube. Let \vec{x} be a set of n points in I and define $G_{\vec{x}}$ to be the discretization of G as described in Section 2. Note that $G_{\vec{x}}$ has the ring like structure of DH, Chord [21] and Viceroy [14]. It is possible to show (and would be shown in the full version of the paper) that $G_{\vec{x}}$ is quite similar to chord. The various algorithms presented for Chord ([21], [12], [5]) apply here as well.

4 Achieving Smoothness

Via our technique many problems reduce to the problem of achieving smoothness in I in a distributed manner. In this section we suggest various algorithms for achieving smoothness. A straightforward algorithm is letting each processor choose its x -value by sampling randomly uniformly a point in I :

Algorithm Single Choice Join:

1. Choose a point $x_i \in [0, 1)$ uniformly at random.
2. Inform all relevant neighbors so that they adjust.

Since the selection of each point is independent from other points, the properties of the algorithm do not change when random processors are deleted.

Lemma 4.1. *After inserting n points there exists w.h.p a segment of length $\Theta(\frac{\log n}{n})$ and a segment of length $\Theta(\frac{1}{n \log n})$. Therefore w.h.p $\rho(\vec{x}) = \Theta(\log^2 n)$.*

Let $s(x_j)$ be the segment that originally covered the point chosen at Step 1. The algorithm could be modified so that x_i is set to be in the middle of $s(x_j)$ dividing it into two. In this case the shortest segment would be of length $\Theta(\frac{1}{n})$ but the longest segment would remain logarithmic. Thus if we insist on \vec{x} being smooth, then a balancing mechanism must be implemented. The idea of the following algorithm, (following the spirit of the two choice paradigm [16]), is to let a joining processor choose *many* locations, and set its x -value to be the best location found.

Multiple Choice Join:

1. Estimate $\log n$.
2. Sample $t \log n$ random points from $(0, 1)$, when t is some constant to be determined later.
3. Check all segments containing those points. Let $s(x_j)$ be the longest of these segments. Set $x_i \leftarrow \frac{x_j + x_{j+1}}{2}$, i.e. divide into 2 the longest segment.

Estimating $\log n$ is a simple task and can be done in various ways. This issue is discussed in Section 6. In the following we assume that an adversary controlled the initial state of the system and we show that the system corrects itself quickly.

Theorem 4.2. *For any initial state, after inserting n points, the largest segment would be of size at most $\frac{2}{n}$.*

Proof. First we prove the following lemma:

Lemma 4.3. *Assume the longest segment is of length $\frac{c}{n}$ (c may be a function of n), then after inserting $\frac{2n}{c}$ points, the longest segment would be of length at most $\frac{c}{2n}$.*

Lemma 4.3. Fix one i such that $|s(x_i)| = \frac{c}{n}$, i.e. $s(x_i)$ is a segment of maximum length. In step (1) of the algorithm $t \log n$ random points are chosen. Whenever one of this point is in $s(x_i)$ we say that $s(x_i)$ is hit. Let X be a random variable that denotes the number of times $s(x_i)$ was hit after $\frac{2n}{c}$ points were inserted. We have

$$E(X) = \frac{c}{n} \cdot \frac{2n}{c} \cdot t \log n = 2t \log n$$

By Chernoff's inequality we have

$$\Pr[X \leq (1 - \delta)E(X)] \leq n^{-\delta^2 t}$$

If we set $\delta^2 t \geq 2$ then by the union bound *all* segment of length $\frac{c}{n}$ were hit at least $(1 - \delta)2t \log n$ times. Now assume that there are k segments of size $\frac{c}{n}$. All in all there were at least $k(1 - \delta)2t \log n$ hits in these segments. Each time a point is inserted there are at most $t \log n$ hits, therefore were at least $2k(1 - \delta)$ points in which a large segment was hit. We have that if $2(1 - \delta) \geq 1$ then all segments of length $\frac{c}{n}$ were split. Both conditions hold when $\delta = \frac{1}{2}$ and $t = 8$. \square

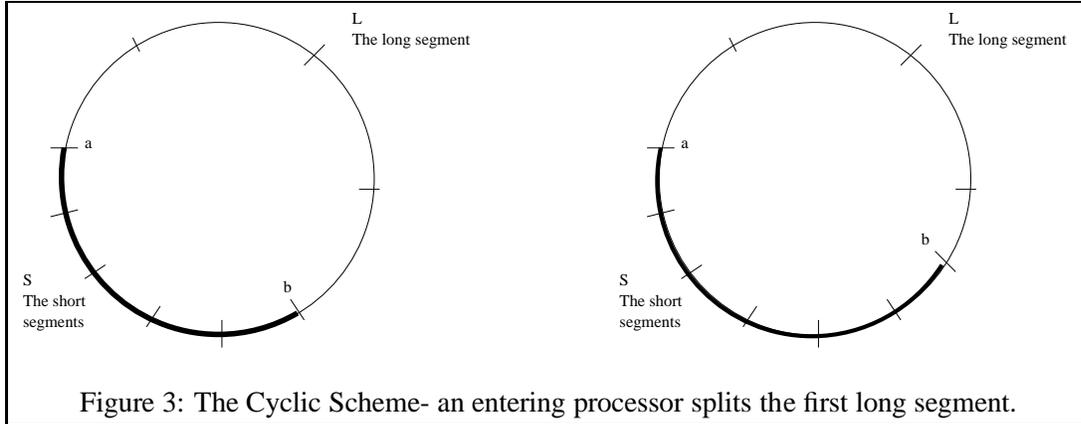
In order to complete the proof we use the previous lemma iteratively $\log c - 1$ times thus inserting n new points and making sure that the length all segments is at most $\frac{2}{n}$. \square

4.1 Handling Deletions

The Multiple Choice Random Selection paradigm guarantees a smooth set of points in the pure Join model; i.e. when we allow processors to join the system but not to leave it. Introducing deletions requires a more complicated scheme. The naive algorithm in which the segment of a deleted processor is taken over by its predecessor on the ring does not suffice. Assume that there are $2n$ processors in I , and randomly delete n of them. With high probability there would be a sequence of $\frac{\log n}{\log \log n}$ consecutive processors that were deleted. We conclude that some balancing mechanism should be implemented whenever a processor is deleted.

The Bucket Solution The 'bucket' solution was suggested in Viceroy [14]. The processors join the system with the Single Choice Join 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 below 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 would be too big or too small.



The correctness of the bucket solution follows from the fact that w.h.p every interval of length $\frac{\log n}{n}$ would contain $\Theta(\log n)$ points (balls and bins). The details of the implementation are more involved but are rather straightforward.

The Cyclic Scheme In the following we sketch a scheme which is deterministic, and can guarantee excellent smoothness even under adversarial deletion and insertion. It takes $\Theta(\log n)$ messages to perform both Join and Leave, and it requires an additional structure. The idea is to divide the segments into long segments and short segment, and to have all the long segments in one *contiguous* interval whose starting point is marked by the point a and its ending by point b . When a processor wishes to enter the system, it does so by splitting into two the long segment adjacent to b . (see Figure 3). When a processor wishes to leave the system, it swaps with the processor in charge of the *first* short segment (whose i.d. in Figure 3 is a) and then leaves creating a long segment. If a processor wishes to leave the system and there is only one long segment in the system, then a slightly more delicate operation should be done which would result with short segments, long segments and at most one medium sized segment whose length is larger than the short ones, but smaller than the long ones. The full description of these procedures would appear in a full version of the paper. The cyclic scheme guarantees a smoothness of 2 even under adversarial deletions and insertions. The main difficulty is finding quickly the processor at the beginning or the end the interval of short segments. This can be done by maintaining an extra structure on top of the DHT. The description of this structure is more involved and is deferred to a full version of the paper.

5 Higher Dimensions

The constructions used so far as an underlying universe a one dimensional line (as did Chord and Viceroy). In some applications it may be useful to apply the same approach to a two dimensional universe. In this case the set of points should be associated with a tessellation of the plain, 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 [18] 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 Distributed Voronoi Diagrams

Definition 4 (planar ordinary Voronoi diagram). Given a set of two or more but a finite number 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* [17] for a thorough overview of Voronoi diagrams and their applications. The Voronoi diagram can be computed by a distributed algorithm, in which every cell is calculated separately. 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. 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, 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 . We say that \vec{x} has *smoothness* ρ if the ratio between the largest and smallest cell is at most ρ and in addition any rectangle $\frac{\rho}{n} \times \frac{\rho}{n}$ in I contains at least one point from \vec{x} .

5.2 Constructing Expanders & Load Balancing Jobs

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 in general and distributed computing in particular. One such application is online load balancing jobs. An algorithm for online job load balancing in a network is presented in [1]. The efficiency of the algorithm depends only on the expansion of the network. Our framework could be used to guarantee that the topology of the network is indeed an expander. An explicit construction for expanders was given by Margulis [15] and later by Gabber and Galil [7]. In this construction a continuous graph G is defined over I by the following two transformations: $f(x, y) = (x + y, y) \bmod 1$, $g(x, y) = (x, x + y) \bmod 1$.

Theorem 5.1 ([7]). *For every set A of points in I such that $\mu(A) \leq \frac{1}{2}$, it holds that $\mu((f(A) \cup g(A)) \setminus A) \geq \frac{(2-\sqrt{3})}{2}\mu(A)$ where $\mu(A)$ is the area of 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 at least $\frac{(2-\sqrt{3})}{2\rho}$. 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, would guarantee a good job load balancing. It is not known yet how to perform routing over these graphs so at the moment this construction can not serve as a basis for a DHT.

6 Fault Tolerant constructions

There are two commonly used methods for modelling the occurrence of faults. The first is the random fault model, in which every processor becomes faulty with some probability and independently from the other processors. The other is the worst case model in which an adversary which knows the state of the system chooses the faulty subset of processors. There are several models that describe the behavior of faulty processors. One of them is the fail-stop model in which a faulty processor is deleted from the system. Another is a spam generating model in which a faulty processor may produce arbitrary false versions of the data item requested. A third model is the Byzantine model in which there are no restrictions over the behavior of faulty processors.

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 processors to be capable of accessing all data items then it is necessary that the degree be at least $\log n$ and that every data item is stored by at least $\log n$ processors. Otherwise with high probability there would be processors disconnected from the system. 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* processors can locate *all* data items. The second routing algorithm guarantees the same but under the random spam generating model. This algorithm has running time (parallel) of $O(\log n)$ and message complexity of $O(\log^2 n)$.

We propose a very simple and easy to implement distributed hash table. Our construction offers logarithmic linkage, load and dilation. It can operate in a highly dynamic environment and is robust against random deletions and random spam generating processors, in the sense that with high probability *all* processors can locate *all* data items.

6.1 Related Work

Several peer-to-peer systems are known to be robust under random deletions ([24], [21], [18]). Stoica *et al* prove that the Chord system [21] is resilient against random faults in the fail-stop model. It does not seem likely that Chord can be made spam resistant without a significant change in its design. Fiat and Saia [20] propose a content addressable network that is robust against deletion and spam in the *worst case* scenario, i.e. when an adversary can choose which processors fail. Clearly in this model some small fraction of the non-failed processors would be denied from accessing some of the data items. Their solution handles a more difficult model than ours, it has several disadvantages. First of all it is not clear whether the system can preserve its qualities when processors join and leave dynamically. Secondly the message complexity is large ($\log^3 n$) and so is the linkage needed ($\log^2 n$). Finally the construction is very complicated and involves a heavy use of randomization. We feel that randomization offers a convenient opportunity for an adversary to diverge from the designated protocol. In a later paper Fiat *et al* [6] solve the first problem yet they do not describe a spam resistant lookup.

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. The difference is in the discretization technique.

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

Property I - The set of points $\vec{x} = x_1, x_2, \dots, x_n$ 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 would not change as long as 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 $i \neq j$, $s(i)$ and $s(j)$ may *overlap*. The point y_i would be updated as processors join and leave the system. The precise manner in which y_i is chosen and updated would be described in the next section.

The edge set of G is defined as follows. A pair of vertices i, j is an edge in G if $s(i)$ and $s(j)$ are connected in G_c or if $s(i)$ and $s(j)$ overlap. The edges of G are anti-parallel. It is convenient to think of G as an undirected graph. A point $a \in I$ is said to be *covered* by i if $a \in s(i)$. We observe the following:

1. Each point in I is covered by $\Theta(\log n)$ processors of G . This means that each data item is stored at $\Theta(\log n)$ processors.
2. Each processor 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 processor i wishes to join the system it does the following:

1. It chooses at random $x_i \in [0, 1)$ ².
2. It calculates a variable q_i which is an estimation of $\frac{\log n}{n}$.
3. It sets $y_i = x_i + q_i \pmod 1$.
4. It 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 processor 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 $\Theta(\frac{\log n}{n})$ would contain $\Theta(\log n)$ points.*

If each processor chooses its x -value uniformly at random from I then property-I holds. Observe that if each processor'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 [14] 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 processor i can easily estimate $\log n$ within a multiplicative factor. Call this estimation $(\log n)_i$. 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* $(\log n)_i$ different x -values.

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

The proof follows directly from Lemma 6.1. Each processor in the system updates its q value and holds an accurate estimation of $\frac{\log n}{n}$ at all times. Therefore property II holds at all times.

Mapping the data items to processors: The mapping of data items to processors is done in the same manner as previously. First data items are mapped into the interval I using a hash function. processor i should hold all data items mapped to points in $s(i)$. The use of consistent hashing [8] is suggested in Chord [21]. Note that all processors holding the same data item are connected to one another so they form a clique. If a processor storing a data item was located, then other processors storing the same data item are quickly located as well. This means that accessing different processors 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* [2]) and thus avoid the need for replication. The data stored by any small subset of the processors would suffice to reconstruct the data item. Weatherspoon and Kubiatowicz [23] suggest that an erasure correcting code may improve significantly the bandwidth and storage used by the system.

²It may be that x_i is chosen by hashing some i.d. of i . In this case it is important that the hash function distribute the x values evenly.

6.3 The Lookup Operation

The lookup procedure emulates a walk in the continuous graph G_c . Assume that processor i wishes to locate data item V and let $v = h(V)$ where h is a hash function, i.e. data item V is stored by every processor which covers the point v . Let $z_i = \frac{x_i + y_i}{2}$ and let σ be the binary representation of z_i . Claim 2.4 states that $\sigma_t(v)$ is within the interval $[z_i - 2^{-t}, z_i + 2^{-t}]$. Conclude that when $t = \log n - \log \log n + O(1)$ it holds that $\sigma_t(v) \in s(i)$. Let t to be the minimum integer for which $\sigma_t(v) \in s(i)$. Call the path between $\sigma_t(v)$ and v the *canonical path* between v and $s(i)$. This gives rise to a natural lookup algorithm. The canonical path exists in G_c , yet by the definition of G , if (a, b) is an edge in G_c , a is covered by i and b is covered by j then the edge (i, j) exists in G . This means that the canonical path can be *emulated* by G .

Simple Lookup: Every point in I is covered by $\Theta(\log n)$ processors. This means that when processor i wishes to pass a message to a processor covering point $z \in I$ it has $\Theta(\log n)$ *different* neighbors that cover z . In the Simple Lookup it chooses *one* of these processors at random and sends the message to it.

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 i is chosen at random from the set of processors and v is chosen at random from I , then the probability a given processor participates in the lookup is $\Theta(\frac{\log n}{n})$.*

Proof Sketch: The proof of statement (1) is immediate. To show the correctness of statement (2) we prove the following: Fix a processor i . The probability processor i participates in the k^{th} step of the routing $1 \leq k \leq \log n$ is $\Theta(\frac{1}{n})$. Summing up over k yields the result. This statement is proved by induction on k . □

Theorem 6.4. *If each processor is deleted 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 processor can locate every data-item.*

Proof. We prove the following claim:

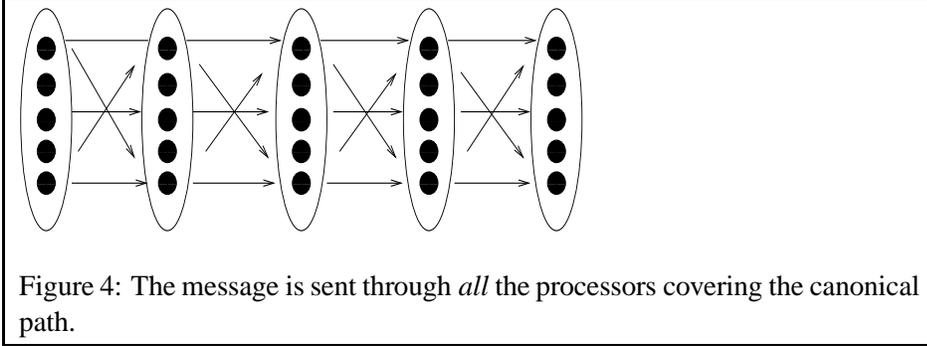
Claim 6.5. *If p is small enough, then w.h.p every point in I is covered by at least one processor.*

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

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

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 processor. 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 processors, and the claim follows. □

For every edge (a, b) in G_c there exists at least one edge in G whose processors 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 processor does not increase significantly. □



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

Theorem 6.6. *The spam resistant lookup has the following properties:*

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

Proof. Statements (2,3) follow directly from the definitions of the spam resistant lookup. Statement (1) follows from the following:

Claim 6.7. *If each processor 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 processors.*

The proof of Claim 6.7 is similar to that of Claim 6.5. Now the proof of Theorem 6.6 is straight forward. It follows by induction on the length of the length of the path. Every point of the canonical is covered by a majority of good processors, therefore every processor along the path would receive a majority of the authentic message. It follows that with high probability *all* processors 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 technique can be used to dynamically construct a graph which embeds *any* family of fixed degree graphs. Theoretically speaking, this result implies that considering scalable systems separately is superfluous; i.e. any problem could be solved in a static environment and then be made 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.

Let (G_1, G_2, \dots) be an infinite family of graphs with degree d . Assume that G_i has 2^i vertices. We show how a smooth set of n points \vec{x} in $[0, 1)$ can be used to construct a graph $G_{\vec{x}}$ that emulates $G_{\lceil \log n \rceil}$. Assume that \vec{x} is smooth and that processor v_i is associated with point x_i . Assume for now that all processors know

n (this assumption would be removed later). Let $l = \lceil \log n \rceil$. We construct $G_{\vec{x}}$ that will embed G_l . Define the function Φ_l from the processors of G_l to the processors of $G_{\vec{x}}$ as follows: $\Phi_l(j) = v_i$ if $\frac{j}{2^l} \in s(x_i)$. The edges of $G_{\vec{x}}$ are defined as follows: $E(G_{\vec{x}}) = \{(v_{i_1}, v_{i_2}) \mid \exists (j_1, j_2) \in E(G_l), \Phi_l(j_1) = v_{i_1}, \Phi_l(j_2) = v_{i_2}\}$. We say that (j_1, j_2) is *simulated* by (v_{i_1}, v_{i_2}) . If \vec{x} is smooth then

1. Every processor in $G_{\vec{x}}$ simulates at most ρ processors in G_l .
2. Every edge in $G_{\vec{x}}$ simulates at most ρ^2 edges in G_l .
3. The degree of $G_{\vec{x}}$ is at most $\rho \cdot d$.

In other words if \vec{x} is smooth then $G_{\vec{x}}$ is a *real time emulation* of G_l . Particularly this means any computation performed by G_l , could be performed by $G_{\vec{x}}$ in constant slow down. see ([13], [10]) for an overview on the literature of real time emulations. It is important to notice that assuming all processors know what n is, each processor can calculate separately which are its neighbors. Next we remove the assumption that all processors know what n is. Smooth \vec{x} implies that each processor v_i can calculate an estimation of n , denoted by n_i , by setting $n_i = \frac{1}{|s(v_i)|}$. By definition $\max_{i,j} \frac{n_i}{n_j} = \rho(\vec{x})$. Therefore it holds that $\forall i \quad \log n_i - \log \rho \leq \log n \leq \log n_i + \log \rho$. If \vec{x} is guaranteed to have smoothness of at most ρ then each processor can calculate a list of length $2 \log \rho$ that contains $\lceil \log n \rceil$. Each processor now would set edges according to every index in its list; i.e. the edges each processor would open would result from the union of the Φ 's on its list.

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

1. *The degree of $G_{\vec{x}}$ is at most $2d \cdot \rho \log \rho$.*
2. *If ρ is a constant then the graph $G_{\vec{x}}$ can emulate in real time the graph $G_{\lceil \log n \rceil}$.*

Acknowledgments: We gratefully thank Kfir Zigdon, Frank Mcsherry, Anna Karlin, Jason Hartline, Cynthia Dwork, Dan Boneh, Frank Dubek, Andrew Goldberg and Dalia Malkhi for their valuable help.

References

- [1] W. Aiello, B. Awerbuch, B. M. Maggs, and S. Rao. Approximate load balancing on dynamic and asynchronous networks. In *ACM Symposium on Theory of Computing*, pages 632–641, 1993.
- [2] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.
- [3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. S., and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX 1996 annual technical conference: January 22–26, San Diego, California, USA*, pages 153–163.
- [4] E. Cohen and H. Kaplan. Balanced-replication algorithms for distribution trees. In *ESA European Symposium on Algorithms*, pages 297–309, 2002.
- [5] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany*, pages 81–86.
- [6] A. Fiat and J. Saia. Censorship resistant peer-to-peer content addressable networks. In *Symposium on Discrete Algorithms (SODA)*, 2002.
- [7] O. Gabber and Z. Galil. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Science*, 22, 1981.
- [8] D. Karger, E. Lehman, F.T. Leighton, R. Panigrahy, M. Levine, and D. 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*, 1997.
- [9] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.

- [10] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44(1):104–147, January 1997.
- [11] F. T. Leighton. *Introduction to parallel algorithms and architectures : arrays, trees, hypercubes*, chapter 3.3. Morgan Kaufmann, San Mateo, CA, 1992.
- [12] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *ACM Conf. on Principles of Distributed Computing (PODC)*, Monterey CA, July 2002.
- [13] B. M. Maggs and E. J. Schwabe. Real-time emulations of bounded-degree networks. *Information Processing Letters*, 66(5):269–276, 1998.
- [14] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *PODC*, 2002.
- [15] G. A. Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, (9(4)), October - December 1973.
- [16] M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices: A survey of the techniques and results, 2000.
- [17] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations — Concepts and Applications of Voronoi Diagrams*. Wiley, Chichester, second edition, 2000.
- [18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc ACM SIGCOMM*, pages 161–172, San Diego CA, August 2001.
- [19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.
- [20] J. Saia, A. Fiat, S. Gribble, A. R. Karlin, and S. Saroiu. Dynamically fault-tolerant content addressable networks. In *First International Workshop on Peer-to-Peer Systems*, MIT Faculty Club, Cambridge, MA, USA, 2002.
- [21] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.
- [22] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2), May 1982.
- [23] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *First International Workshop on Peer-to-Peer Systems*, MIT Faculty Club, Cambridge, MA, USA, 2002.
- [24] B.Y. Zhao and J. 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.