Butterflies and Peer-to-Peer Networks

Mayur Datar

Department of Computer Science, Stanford University, Stanford, CA 94305, USA
datar@cs.stanford.edu

Abstract. Research in Peer-to-peer systems has focussed on building efficient Content Addressable Networks (CANs), which are essentially distributed hash tables (DHT) that support location of resources based on unique keys. While most proposed schemes are robust to a large number of random faults, there are very few schemes that are robust to a large number of adversarial faults. In a recent paper ([1]) Fiat and Saia have proposed such a solution that is robust to adversarial faults.
We propose a new solution based on multi-butterflies that improves upon the previous solution by Fiat and Saia. Our new network, *multi-hypercube*, is a fault tolerant version of the hypercube, and may find applications to other problems as well. We also demonstrate how this network can be maintained dynamically. This addresses the first open problem in the paper ([1]) by Fiat and Saia.

1 Introduction

Peer-to-peer (P2P) systems are distributed systems without (ideally) any centralized control or hierarchical organization, which make it possible to share various resources like music [7, 3], storage [5] etc over the Internet. One approach to building P2P systems is to build a distributed hash table (DHT) that supports location of resources based on their unique key. Such a network is called a content addressable network (CAN) and various solutions ([8, 9, 11]) have been proposed for building efficient CANs.

While most schemes for building CANs are fairly robust against random attacks, a powerful agent like a government or a corporate can attack the system by carefully deleting (making faulty) chosen points or nodes in the system. For instance, the Gnutella [3] file sharing system, while specifically designed to avoid the vulnerability of a central server, has been found (refer [10]) to be highly vulnerable to an attack by removing a very small number of carefully chosen nodes. Thus it is unclear if any of these systems are robust against massive orchestrated attacks.

Recent work by Fiat and Saia [1] presents a CAN with n nodes that is censorship resistant, i.e. fault tolerant to an adversary deleting up to a constant fraction[1] of the nodes. It is clearly desirable for a P2P system to be censorship resistant and to the best of our knowledge this is the first such scheme of its kind.

[1] The paper provides a system that is robust to deletion of up to half the nodes by an adversary. It can be generalized to work for arbitrary fraction.

However a drawback of the solution presented in [1] is that it is designed for a fixed value of n (the number of participating nodes) and does not provide for the system to adapt dynamically as n changes. In fact the first open problem that they mention in their paper (Sect. 6 of [1]) is the following: "*Is there a mechanism for dynamically maintaining our network when large numbers of nodes are deleted or added to the network? ..*"

This paper solves this open problem by proposing a new network that can be maintained dynamically and is censorship resistant. Our new network, *multi-hypercube*, is a fault tolerant version of the hypercube network and may find applications to other problems as well. We first present a static solution that is much simpler than that presented in [1] and improves upon their solution. Next we show how we can dynamically maintain our network as nodes join and leave.

A drawback of our solution is that it requires a reconfiguration after an adversarial attack (details in Sect. 3.1). This does not involve adding new edges or nodes to the network. It only involves sending messages along existing edges to label some nodes as "faulty". This reconfiguration step requires $O(\log n)$ time and $O(n \log n)$ messages are sent, where n is the number of nodes in the network.

We present a table (refer to Table 1) that compares all of these solutions based on some important factors. The parameter n is the number of nodes participating in the network.

Network	linkage (degree)	query cost (path length)	Messages per query	Fault Tolerance	Dynamic	Data Replication factor
CAN [9]	$O(d)$	$O(n^{1/d})$	$O(n^{1/d})$	?	Yes	$O(1)$
Chord [11]	$O(\log n)$	$O(\log n)$	$O(\log n)$	Random	Yes	$O(1)$
Viceroy [8]	7	$O(\log n)$	$O(\log n)$	?	Yes	$O(1)$
CRN [1]	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	Adversarial	No	$O(\log n)$
MBN (this paper)	$O(\log n)$	$O(\log n)$	$O(\log n)$	Adversarial	No	$O(1)$
DMBN (this paper)	$O(\log n)$	$O(\log n)$	$O(\log n)$	Adversarial	Yes	$O(1)$

Table 1. Comparison of recent solutions.

Paper Organization: We begin by briefly reviewing some of the related work in Sect. 2. In Sect. 3 we present a simpler and better censorship resistant network. Section 4 provides a dynamic construction of our network. Finally we conclude with a discussion of open problems in Sect. 5.

2 Related Work

Most CANs are built as an overlay network. The goal is to build a CAN with short query path length since it is directly related to the latency observed by the node that issues the query. Besides a small query path length, other desirable

features of a solution include low degree for every node, fewer messages per query, fault tolerance etc.

Recently various solutions – *CAN* ([9]), *Chord* ([11]), *Viceroy* ([8]) etc. – have been proposed to building CANs. Please refer to Table 1 for a comparison of their various performance parameters. A common feature to all of these solutions is the use of an underlying abstract hash space to which nodes and data items are hashed. It is critical to all these schemes that the data items are hashed uniformly and deterministically based on their keys, so that any node can compute the hash value of a data item solely based its unique key.

The *CAN* system designed by Ratnasamy, Francis et al [9] uses a virtual d-dimensional (for a fixed d) Cartesian coordinate space on a d-torus as its hash space. The hash space used by *Chord* [11] and *Viceroy* [8] is identical. It can be viewed as a unit circle $[0,1)^2$ where numbers are increasing in the clockwise direction. The *Chord* system tries to maintain an approximate hypercube network in a dynamic manner, while the *Viceroy* system tries to maintain an approximate butterfly network in a dynamic and decentralized manner.

The censorship resistant network (henceforth CRN) developed by Fiat and Saia [1] departs from all of the above solutions in trying to provide adversarial fault tolerance. The aim is to build a network such that even after an adversary deletes (makes faulty) up to $n/2$ nodes, $(1-\epsilon)$ fraction of the remaining nodes should have access to $(1-\epsilon)$ fraction of the data items, where ϵ is a fixed error parameter. However, their solution assumes that there are n nodes participating in the network, where n is fixed. Their solution can be extended to form a network that is *spam resistant*, i.e. resistant to an adversary that can not only delete a large number of nodes but also make them collude so that they forward arbitrary false data items (or messages) during query routing. In a very recent paper ([2]), the authors have extended their previous work to build a CAN that is *dynamically fault-tolerant*. Their notion of *dynamic* differs from ours and as per their notion of *dynamic fault-tolerance*, the network is built for a certain value n of the number of participating nodes. However, there may be a large turn around of the participating nodes and during any time period for which the adversary deletes γn nodes, δn ($\delta > \gamma$) new nodes are added to the system, always maintaining that there are at least κn live nodes for some fraction κ. At every time instant the network should remain censorship resistant. They build such a system that is an adaptation of CRN and has properties similar to it.

3 Multi-Butterfly Network (Multi-Hypercube)

In this section we present a censorship resistant network based on multi-butterflies, which we refer to as MBN (Multi-Butterfly Network). Our solution is better than CRN in the following respects:

1. While routing in CRN requires $O(\log^2 n)$ messages, routing in our network requires $O(\log n)$ messages.

[2] While the *Chord* paper [11] describes their hash space as "identifier circle modulo 2^m" the two are equivalent

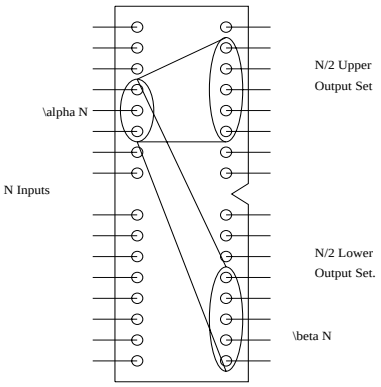

Fig. 1. Splitter with N inputs and N outputs.

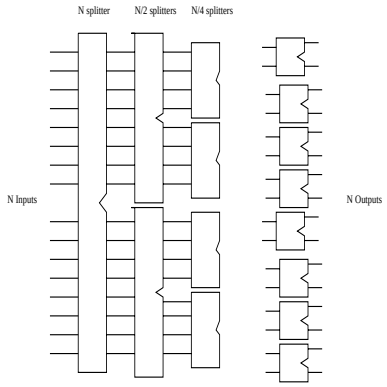

Fig. 2. A splitter network with n rows, $\log n + 1$ levels.

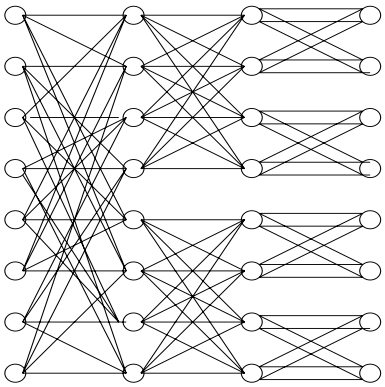

Fig. 3. Twin Butterfly with 8 inputs.

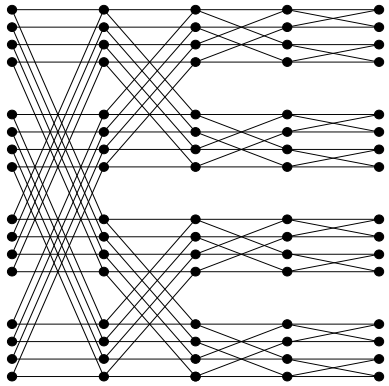

Fig. 4. Butterfly network of 16 rows and 5 levels.

2. The data replication factor in CRN is $O(\log n)$. i.e. every data item is stored at $O(\log n)$ nodes. In our network the data replication factor is $O(1)$.
3. The data availability in our network degrades smoothly with the number of adversarial deletions. No such guarantees are given for CRN.

Similar to [1] we first present a static version, where the number of participating nodes(n) is fixed. Later, we will provide a dynamic construction that maintains an *approximate version* of this network and has similar properties.

Our construction is based on multi-butterflies. Fig. 4 shows a twin-butterfly. Multi-butterfly networks were introduced by Upfal [12] for efficient routing of permutations and were later studied by Leighton and Maggs [6] for their fault tolerance. Please refer to their papers for details. Butterflies and Multi-butterflies belong to the class of splitter networks (Fig. 3), whose building block is a splitter (Fig. 2). In an N-input splitter of a multi-butterfly with multiplicity d, every

input node is connected to d nodes from the upper and lower output nodes. Similarly every output node has edges from $2d$ input nodes. The splitter is said to have (α, β)-expansion if every set of $k \leq \alpha N$ input nodes is connected to at least βk upper output nodes and βk lower output nodes, where $\alpha > 0$ and $\beta > 1$ are fixed constants (Refer to Fig. 2). Thus, the N input nodes and $N/2$ upper (lower) output nodes form a concentrator with (α, β)-expansion. A multi-butterfly is said to have (α, β)-expansion if all its splitters have (α, β)-expansion. Splitters are known to exist for any $d \geq 3$, and they can be constructed deterministically in polynomial time [12], but randomized wirings will typically provide the best possible expansion. In fact, there exists an explicit construction of a splitter with N inputs and any $d = p + 1$, p prime, and $\beta \leq d/(2(d-4)\alpha + 8)$ (Corollary 2.1 in [12]). In the case of a butterfly network any pair of input and output nodes are connected by a unique (bit correcting) logical path. However, in the case of a multi-butterfly there is a lot of redundancy since there is a choice of d edges to choose from, at every node, instead of a single edge as in the case of a butterfly. As a result there is a myriad of paths that connect any pair of input and output nodes. This redundancy is the key to the fault tolerance of a multi-butterfly, which was formally proven in [6].

At an intuitive level, our aim is to build a network such that even after an adversary deletes a constant fraction of the nodes in the network, $\Omega(n)$ remaining nodes are each connected by small length paths to $\Omega(n)$ of remaining nodes. In other words even after an adversary deletes a constant fraction of the nodes, there should remain a connected component of size $\Omega(n)$ and small diameter. While this is not the end goal of a censorship resistant network, we will see later how such a network can be easily enhanced to make it censorship resistant. Although, nodes in a multi-butterfly have constant degree, a multi-butterfly with n nodes can only tolerate $O(n/\log n)$ faults and is not suited for Censorship Resistance. Hence we build a new network called *multi-hypercube* that is based on multi-butterflies and is in fact the fault tolerant version of hypercube. In short, a multi-hypercube is to a multi-butterfly as a hypercube is to a butterfly. If the role of all the nodes in a single row of a multi-butterfly is played by a single node then what we get is a multi-hypercube[3]. Consider an N input splitter in a multi-hypercube. In this splitter, upper $N/2$ input nodes are connected to lower $N/2$ output nodes (and vice versa), via an expander of degree d. As a result we get better expansion factor (β) for the same degree as compared to that in a multi-butterfly, where instead of a $(N/2, N/2)$ expander we have a $(N, N/2)$ concentrator. To the best of our knowledge this network has not been studied earlier, neither are we aware of the use of the term multi-hypercube. A formal definition follows:

Multi-Hypercube: A multi-hypercube of dimension m and multiplicity d consists of 2^m nodes, where every node has degree $2md$. A node with binary representation $b_1 b_2 \ldots b_m$ is adjacent to $2d$ nodes at each level i $(1 \leq i \leq m)$. At level i it has "out-edges" with d nodes whose first i bits are $b_1 b_2 \ldots b_{i-1} \overline{b_i}$. It also has

[3] The caveat is that in every splitter we only maintain the "cross" edges and not the "straight" edges

"in-edges" from d nodes belonging to the same set, i.e. with first i bits given by $b_1 b_2 \ldots b_{i-1}\overline{b_i}$. The connections are such that the expansion property holds for every splitter, like in the case of a multi-butterfly.

Thus, a multi-hypercube with n nodes has degree $2d \log n$ for each node. A multi-hypercube is a fault tolerant version of the hypercube network, and turns out to be ideal for censorship resistance. We hope that this network will find other applications as well.

Given n, the network that we build is a multi-hypercube with n nodes and (α, β) expansion. The fault tolerance property that we will prove (Theorem 1) about the multi-hypercube is the exact equivalent of the corresponding property for a multi-butterfly. We refer to this network as the Multi-butterfly network (MBN), since we prefer to visualize it as a multi-butterfly. Data items are randomly hashed onto any node. Thus the data replication factor is 1 and using consistent hashing, as in [11], we can guarantee that whp the load on any node is at most $O(\log n)$ times the expected average load.

Distributed creation: Similar to [1] we describe how we create our network in a distributed manner. In the first round every node broadcasts its unique identifier (ip-address) to all other nodes. Based on the identifiers that a node receives from other nodes it determines its index i in the sorted list of identifiers. This can be done by comparing every identifier with its own and maintaining a count of smaller identifiers. Thus at the end of round one every node knows its index i in the sorted list. Based on its index i every node computes the indexes of all the other nodes that it will connect to. Note, every node connects to at most $2d \log n$ other nodes. In round two every node broadcasts its index (i) and identifier (ip-address) to all other nodes. Every node in turn remembers the identifiers of the pertinent $2d \log n$ nodes and forms a connection with them. As mentioned before data items are randomly hashed onto any one of the n nodes based on their key and this hash function is known to all the nodes. Data items can be inserted by performing a query on them to reach the node they must belong to and then inserting them at that node. The construction of the network requires 2 broadcasts from every node, with a total of $2n^2$ messages and assumes that each node has $O(\log n)$ memory, similar to the creation of CRN.

Routing: Every node (source) that wishes to access a data item computes the hash of its key and finds out the index of the node the data item belongs to (destination). Routing between the source and destination is done using the standard, logical *bit-correcting* path as in a hypercube. Due to the redundancy in connections, in a fault free multi-hypercube we will have a choice of d out edges at each level (splitter) of the routing. This choice may be reduced if some of the nodes become faulty, as we shall see later. The number of messages sent for a single query is at most $\log n$ and time taken is also $\log n$.

3.1 Fault Tolerance

In this subsection we prove the fault tolerance for our network. The proof is similar to that presented in [6]. We will view the multi-hypercube as a multi-butterfly where a single node plays the role of all the nodes in a row of the multi-

butterfly. In the discussion below we will refer to nodes on level 0 (leftmost level in the figures) as input nodes and nodes on level $\log n$ as output nodes treating them separately. But in reality same node is playing the role of all the nodes in a row. We prove the following theorem:

Theorem 1. *No matter which f nodes are made faulty in the network, there are at least $n - \frac{\beta f}{\beta - 1}$ nodes that still have a $\log n$ length logical path to at least $n - \frac{f}{\alpha(\beta-1)}$ nodes, such that all nodes on the path are not faulty, where (α, β) are the expansion parameters for every splitter.*

We first describe which outputs to remove. Examine each splitter in the multi-butterfly and check if more than $\epsilon_0 = \alpha(\beta - 1)$ fraction of the input nodes are faulty. If so, then "erase" the splitter from the network as well as all descendants nodes, i.e all nodes to the right of the splitter. The erasure of an m-input splitter causes the removal of m multi-butterfly outputs, and accounts for at least $\epsilon_0 m$ faults. Moreover, since a (faulty) node plays the role of all nodes in the same row the output nodes "erased" by it, by virtue of it being in different splitters, are the same. Thus we can attribute the erasure of an output node to a unique largest splitter that "erased" it. Hence, at most $\frac{f}{\epsilon_0} = \frac{f}{\alpha(\beta-1)}$ outputs are removed by this process. We next describe which inputs to remove. Remember, for every splitter each input node is connected to d nodes from either upper or lower outputs depending on its position in the splitter. Working from the $\log n$th level backwards, examine each node to see if all of its outputs lead to faulty nodes that have not been erased. If so, then declare the node as faulty. We prove that at most $f/(\beta - 1)$ additional nodes are declared to be faulty at each level of this process.

Lemma 1. *In any splitter, at most α fraction of the inputs are declared to be faulty as a consequence of propagating faults backward. Moreover, at most $\alpha/2$ fraction are propagated by faulty upper outputs and at most $\alpha/2$ fraction by faulty lower outputs.*

Proof. The proof is by induction on the level, starting at level $\log n$ and working backwards. The base case at level $\log n$ is trivial since there are no propagated faults at this level. Now consider an arbitrary m-input splitter. If a splitter contains more than $\frac{\alpha m}{2}$ propagated faults from its upper outputs, then these faults must have originated from faults in upper outputs and, in addition, the upper outputs could not have been erased. Consider the set U of faulty upper outputs (propagated or otherwise) that led to the propagated faults in the input. Since each propagated input fault is connected d upper output faults, we conclude that $|U| > \alpha\beta m/2$ (using the expansion property). By induction hypothesis, and the fact that the upper outputs were not erased (and hence had less than $\frac{\epsilon_0 m}{2}$ faults), we know that $|U| < \frac{\alpha m}{2} + \frac{\epsilon_0 m}{2} = \alpha\beta m/2$ which is a contradiction. Hence there could not have been more than $\frac{\alpha m}{2}$ ($\alpha/2$ fraction) propagated faults to the inputs from faulty upper(lower) outputs.

Lemma 2. *Even if we allow the adversary to make f nodes faulty on every level there will be at most $\frac{f}{\beta-1}$ propagated faults on any level.*

Proof. The proof is again by induction on level. Consider some level l and assume that it has more than $\frac{f}{\beta-1}$ propagated faults. These faults are divided among input nodes of splitters linking level l to level $l+1$. By previous Lemma, we know that for every splitter with m inputs, there are at most $\alpha m/2$ propagated faults to the upper inputs and at most $\alpha m/2$ propagated faults to the lower inputs. Hence we can apply the expansion property to each splitter. Hence there must be more than $\frac{\beta f}{\beta-1}$ faults on level $l+1$. This is a contradiction however, since level $l+1$ can have at most $f + \frac{f}{\beta-1} = \frac{\beta f}{\beta-1}$ total faults by induction. Hence, level l can have at most $\frac{f}{\beta-1}$ propagated faults.

We erase all the remaining faulty nodes. The process of labeling nodes faulty guarantees that an input node that is not faulty has a path to all the output nodes that are not erased. This leaves a network with $n - \frac{\beta f}{\beta-1}$ input nodes and $n - \frac{f}{\alpha(\beta-1)}$ outputs nodes such that every remaining input has a logical path to every remaining output. The process of marking nodes "faulty" is important to the efficient functioning of a CAN after an adversarial attack. It tells every node the set of nodes it should not forward a query and restricts the earlier choice of d nodes it had at each level. While the algorithm above gives an off-line (centralized) algorithm to label nodes faulty what we require is an online algorithm that lets us do this without requiring a central authority. It was shown in [4] that such an algorithm exists. In other words we can reconfigure a faulty network in an online manner with just the live nodes talking to each other. This reconfiguration step requires $O(\log n)$ time and $O(n \log n)$ messages are sent. Please refer to [4] for details. It is required that we do this reconfiguration after an adversarial attack. Since we cannot figure out when the attack has occurred, we suggest that the network does this reconfiguration at regular intervals. During the time interval after an adversary attack and before the reconfiguration is done, it may happen that a node may forward a query to another node that is "faulty", but not yet so marked, and consequently the query may not reach the destination. In such a case the node can retry forwarding the query through another node hoping that it is not faulty, after waiting for a "timeout". However such retry's can take a lot of time and we may end up sending a lot of messages. This temporary lack in query routing efficiency of the network should be contrasted with the advantage that it has of requiring fewer messages per query.

One possible choice of parameters we may choose for our network are as follows: Choose the multiplicity d such that $\alpha(\beta-1) \geq 2/3$ and $\beta \geq 3$. Substituting these values in the Thm. 1 gives us that no matter which f nodes are made faulty, there are at least $n - \frac{3f}{2}$ nodes that can each reach $n - \frac{3f}{2}$ nodes through a logical path. Note that the guarantee above is deterministic as opposed to whp, as in case of CRN([1]). Moreover we can characterize the "loss" smoothly in our case. Thus if we loose $f = \sqrt{n}$ nodes we know that all but $n - \frac{3\sqrt{n}}{2}$ nodes can reach all but $n - \frac{3\sqrt{n}}{2}$ nodes. Such guarantees are not given in the case of CRN, which does not characterize the behavior when the number of faults is sublinear.

Enhancement: Theorem 1 guarantees that no matter which $n/2$ nodes are made faulty by the adversary, there are $n/4$ remaining nodes (call this set I) that

are each connected by $\log n$ length logical paths to $n/4$ nodes (call this set O). However, this by itself is not sufficient to give us a censorship resistant network. We achieve censorship resistance by further enhancing the network as follows: Every node in the network is additionally connected to $k_1(\epsilon)$ (a constant that is function of the error parameter ϵ) random nodes in the network. Moreover, every data item is maintained at $k_2(\epsilon)$ nodes in the network, which are specified by $k_2(\epsilon)$ independent hash functions. First step increases the degree of every node by an additive constant, while the second step makes the data replication factor $k_2(\epsilon)$ (a constant) instead of 1.

Routing takes place as follows: A node x that is looking for a data item y computes the $k_2(\epsilon)$ nodes (call this set O') that the data item will be maintained at, using to the different hash functions. Let I' denote the set of $k_1(\epsilon)$ random nodes that the node x is connected to as above. For every pair of nodes $(a, b) \in I' \times O'$, from the cross product of I' and O', the node uses a as a proxy to route the query to b. Note that the cross product $I' \times O'$ has a constant size. Using Lemma 4.1 from [1] it is easy to see that for all but ϵ fraction of the nodes the set I' contains a node from I, i.e. $I' \bigcap I \neq \phi$. Similarly all but ϵ fraction of the data items are maintained at some node in O, i.e $O' \bigcap O \neq \phi$. The above two conditions guarantee a successful query. However, the guarantees in this extension are not deterministic. We can summarize the properties of the network in the following theorem.

Theorem 2. *For a fixed number of participating nodes n, we can build a MBN such that:*

- *Every node has indegree and outdegree equal to $d \log n$.*
- *The data replication factor is $O(1)$.*
- *Query routing requires no more than $\log n$ hops and no more than $\log n$ messages are sent.*
- *Even if f nodes are deleted (made faulty) by any adversary at least $n - \frac{3f}{2}$ nodes can still reach at least $n - \frac{3f}{2}$ nodes using $\log n$ length paths.*
- *This network can be enhanced so that as long as the number of faults is less than $n/2$, whp $(1-\epsilon)$ fraction of the live nodes can access $(1-\epsilon)$ fraction of the data items.*

4 Dynamic Multi-Butterfly Network

In this section we will describe how to dynamically maintain the MBN described in the earlier section in an "approximate" manner, as n changes over time. The network that we build has the following properties:

- Every node will be connected to $O(\log n)$ other nodes. Query requires $O(\log n)$ time and $O(\log n)$ messages are sent during each query.
- The fault tolerance of the network will be similar to that of MBN. Namely, if at any time there are f adversarial faults, $n - O(f)$ nodes still have $O(\log n)$ length path to $n - O(f)$ of the nodes.

– We assume that there are no adversarial faults while the network builds. While at every instance the network that is built is fault tolerant to adversarial faults, we cannot add more nodes to the network once adversarial faults happen. In other words our network admits only one round of an adversarial attack. We do however allow random faults as the network builds.

We refer to our dynamic network as DMBN for Dynamic Multi-Butterfly Network. Similar to *Chord* [11] and *Viceroy* [8] we hash the nodes and data items onto a unit circle $[0, 1)$ using their ip-address, keys etc. We refer to the hash value as the identifier for the node or data item. We assume that the precision of hashing is large enough to avoid collisions. For $x \in [0, 1)$, $Successor(x)$ is defined as the node whose identifier is clockwise closest to x. A data item with identifier y is maintained at the node $Successor(y)$. We also maintain successor and predecessor edges similar to *Chord* and *Viceroy*. In these respects our network is exactly similar to *Chord*. While *Chord* tries to maintain an approximate hypercube we try to maintain an approximate multi-hypercube. In order to do so we need to define an appropriate notion of splitters and levels. Consider a dyadic interval $I = [z, z + 1/2^i)$ $(i \geq 0)$. This interval is further broken into two intervals $I_l = [z, z + 1/2^{i+1}), I_u = z + 1/2^{i+1}, z + 1/2^i)$. Let $S = S_l \bigcup S_u$ be the set of nodes whose identifiers belong to the intervals I, I_l, I_u respectively . The sets of nodes S_u, S_l along with the edges between them form a splitter in DMBN. As in MBN, all nodes in S_u, maintain outgoing edges with d random nodes from S_l and vice versa. The index i that determines the width of the dyadic interval defines the level to which this splitter belongs.

4.1 Definitions and Preliminaries

We will refer to a node with identifier x as node x. Let $x = 0.x_1x_2x_3 \ldots x_p$ be the binary representation of x, where p is the precision length.

Definition 1. *A dyadic interval pair (DIP) for x $(0 \leq x < 1)$ at level i $(i \geq 0)$ is defined as* $DIP(x, i) = \{[0.x_1x_2 \ldots x_i, 0.x_1x_2 \ldots x_i1 = 0.x_1x_2 \ldots x_i + 1/2^{i+1}), [0.x_1x_2 \ldots x_i1, 0.x_1x_2 \ldots x_i + 1/2^i)\}$.

Thus $DIP(x, i)$ are two consecutive intervals of length $1/2^{i+1}$ which agree on x on the first i bits. The nodes that belong to $DIP(x, i)$ form a level i splitter of the multi-hypercube that we are trying to maintain in an approximate manner. Every node in this interval pair maintains edges with d random nodes from the other interval in the pair. This other interval is defined below.

Definition 2. *A dyadic interval (DI) for x $(0 \leq x < 1)$ at level i $(i \geq 0)$ is defined as* $DI(x, i) = [0.x_1x_2 \ldots x_i\overline{x_{i+1}}, 0.x_1x_2 \ldots x_i\overline{x_{i+1}} + 1/2^{i+1})$.

Lemma 3. *For any x $(0 \leq x < 1)$ and $i \geq 0$, let k_u and k_l be the number of nodes whose identifiers belong to the 2 intervals in the dyadic interval pair $DIP(x, i)$. If $k = k_u + k_l \geq c \log n$ for some constant c, then with probability at least $1 - 1/n^2$, $max(\frac{k_u}{k_l}, \frac{k_l}{k_u}) < 2$.*

Proof. Observe that any node whose identifier belongs to the interval pair $DIP(x,i)$ has an equal probability of getting hashed onto any of the two intervals, i.e. $E(k_u) = E(k_l) = k/2$. It follows from a trivial application of Chernoff bounds that if k is large enough ($c \log n$), then whp (at least $1 - 1/n^2$), k_l, k_u will not be off by a factor more than 2.

The Lemma says that if the dyadic interval pair is fairly populated it will be evenly balanced, up to a factor 2.

Every node x can get a crude estimate of the number of nodes in the system as follows: Let $n_0 = 1/d(x, successor(x))$, where $d(x, successor(x))$ is the distance between x and its successor. The following lemma about this estimate is taken from [8](Lemma 4.3). It shows that every node can estimate $\log n$ within a constant factor, whp.

Lemma 4. *Let the system consist of n servers (nodes) whose identities (hash values) are randomly distributed on the unit circle. Then w.h.p. we have that for all nodes, the estimate n_0 satisfies $\frac{n-1}{2 \log n} \leq n_o \leq n^3$.*

4.2 Dynamic Construction

Node Join: Similar to *Chord* [11] a typical query in our network is of the form $successor(x)$ where given a value x we return the node $successor(x)$. A node that joins the network has access to some live node in the network, which it will use to issue *successor* queries as it joins the network. The following steps are executed by every new node with identifier x that joins the network:

- Similar to *Chord* it finds $successor(x)$ and establishes edges to successor and predecessor nodes.
- Similar to *Chord*, all data items that are currently held by $successor(x)$, but have identifiers less than x are transferred to x.
- $i = 0, done = false$
  do
  - Check if the total number of nodes in the interval $DI(x,i)$ exceeds $c \log n$ for some constant c (based on Lemma 3). This can be easily done using a single successor query and then following successor edges till we encounter $c \log n$ nodes or overshoot the interval.
  - If there are less than $c \log n$ nodes in $DI(x,i)$ connect to all of them, set $done = true$.
  - Else choose d random values $(r_1^i, r_2^i, \ldots, r_d^i)$ from the interval $DI(x,i)$. For every value z in this set issue the query $successor(z)$ and maintain an edge with this node.
  - Increment i (Move to the next level splitter)

  while(!$done$)

In short every node establishes edges with d random nodes from the dyadic interval $DI(x,i)$ for $i \geq 0$. It does so till the interval $DI(x,i)$ becomes so small (as i increases)that it contains only $c \log n$ nodes, at that point it maintains a

connection to all of them. It is easy to prove that in less than $O(\log n)$ levels the number of nodes that fall into a dyadic interval reduce to $O(1)$ whp. As a result every node will maintain connections to $O(\log n)$ dyadic intervals and have degree $O(\log n)$.

Continuous Update: We will describe in short how these edges are maintained (updated) over time as nodes leave and join. Choosing d random values $(r_1^i, r_2^i, \ldots, r_d^i)$ and maintaining connection with $successor(r_i^k)$ $(1 \leq k \leq d)$ is a mechanism to guarantee that every node x maintains connection with d random nodes from $DI(x,i)$. As nodes join and leave $successor(r_k^i)$ may change. As a result it is necessary for nodes to check if the node that it maintains an edge to is indeed $successor(r_k^i)$. It is sufficient to do this check whenever the number of nodes reduce or grow by a factor 2, i.e. whenever a nodes estimate of $\log n$ changes. A node should also check for other boundary conditions like "does the smallest dyadic interval it maintains edges to have less than $c \log n$ nodes". We could also follow a more pro-active strategy such that whenever a node leaves or joins the network we adjust the connections corresponding to r_i^k's for other nodes. We omit the details for lack of space.

Node Leave: Similar to *Chord* a node that wishes to leave transfers its data items to its successor.

4.3 Routing:

Routing takes place along the usual *bit-correcting* logical path. At every level i the ith bit of the query y (data identifier) is compared with the ith bit of the node x. If the two bits match the query enters the next level. Else it is forwarded to one of the d random nodes that x connects to from $DI(x,i)$, thereby "correcting" the ith bit. Finally, as i increases the interval $DI(x,i)$ becomes so small that it has only $c \log n$ nodes, at which point x connects to all the nodes in that interval and can forward the query to $successor(y)$.

4.4 Fault Tolerance:

Consider a dyadic interval pair A, B that forms a splitter at some level i and has more than $c \log n$ nodes[4]. We know, from Lemma 3, that the interval pair is well balanced (up to a factor 2) whp. In our construction we maintain that every node in A connects to d random edges in B and vice versa. It follows from Lemma 4.1 in [1] that whp (at least $1 - 1/n^2$), the splitter formed by the interval pair A, B will have the crucial expansion property for parameters α, β that satisfy $2\alpha\beta < 1$ [5] The proof for fault tolerance follows exactly as in MBN. We replace splitters with the dyadic intervals and the arguments follow. We have to be slightly careful in our argument due to the slight imbalance in the number of nodes in a dyadic interval pair (Lemma 3). We omit the details due to lack

[4] If the number of nodes is less than $c \log n$ a complete bipartite graph is maintained between A, B.

[5] Factor 2 comes the slight imbalance in the number of nodes in the two intervals.

of space. We get the following theorem, which is the equivalent of Theorem 1. Note, the guarantees are no more deterministic but instead probabilistic. The phrase *whp* in the theorem is with respect to the random hashing of nodes and the random connections maintained by nodes in different splitters.

Theorem 3. *No matter which f nodes are made faulty in the network, there are at least $n - \frac{\beta f}{\beta - 1}$ nodes that still have a $O(\log n)$ length path to at least $n - \frac{f}{\alpha(\frac{\beta}{2} - 1)}$ nodes whp.*

Similar to the static case(MBN), DMBN must reorganize itself after an adversary attack. In order to make the network censorship resistant we hash the data items multiple times ($k_2(\epsilon)$) and have every node connect to extra $k_1(\epsilon)$ nodes, similar to the enhancements discussed in Sect. 3.1.

It is important to note that the dynamic construction wont work after an adversarial attack. The construction assumes that all $successor(x)$ queries will be answered correctly. This is necessary for new nodes to establish their connections. However once an adversarial attack has taken place such a guarantee cannot be given. After the attack remaining nodes can still query for data and they are guaranteed to have access to most of the data. This follows from the fact that we maintained a fault tolerant network till the time of the attack.

5 Open Problems

Some open problems that remain to be addressed for fault tolerant CANs are:

- Can we build an "efficient" dynamic CAN that is fault tolerant to adversarial faults and allows dynamic maintenance even after an adversary attack, i.e. allows multiple rounds of adversary attack.
- Could multi-butterflies be used in an efficient manner to construct a spam resistant network.
- Are there lower bounds for average degree of nodes, query path length etc. for a network that is fault tolerant to linear number of adversarial faults.

Acknowledgment: The author would like to thank the anonymous referee for various helpful suggestions.

References

1. A. Fiat, and J. Saia. Censorship Resistant Peer-to-Peer Content Addressable Network. In *Proc. Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2002.
2. J. Saia, A. Fiat, S. Gribble, A. Karlin, and S. Saroiu. Dynamically Fault-Tolerant Content Addressable Networks. In *First International Workshop on Peer-to-Peer Systems*, 2002.
3. Gnutella website. `http://gnutella.wego.com/`.

4. A. Goldberg, B. Maggs, and S. Plotkin. A parallel algorithm for reconfiguring a multi-butterfly network with faulty switches. In *IEEE Transactions on Computers*, 43(3), pp. 321-326, March 1994.
5. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, and S. Rhea. OceanStore: An architecture for global-scale persistent storage. In *Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2000), Boston, MA, USA, November 2000.*
6. T. Leighton, and B. Maggs. Expanders Might be Practical: Fast Algorithms for Routing Around Faults on Multibutterflies. In *Proc. of 30th IEEE Symposium on Foundations of Computer Science, pp. 384-389, Los Alamitos, USA. 1989.*
7. Napster website. `http://www.napster.com/`.
8. D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Lookup Network. In *Proc. of the 21st ACM Symposium on Principles of Distributed Computing (PODC 2002), Monterey, CA, USA, July 2002.*
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM 2001 Technical Conference, San Diego, CA, USA, August 2001.*
10. S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. of Multimedia Computing and Networking, 2002.*
11. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM 2001 Technical Conference* August 2001.
12. E. Upfal. An $O(\log N)$ deterministic packet-routing scheme. In *Journal of ACM, Vol. 39, No. 1, Jan. 1992, pp 55-70.*