Butterflies and Peer-to-Peer Networks

Mayur Datar*

Abstract

The popularity of systems like Napster, Gnutella etc. have spurred recent interest in Peer-to-peer systems. A central problem in all these systems is efficient location of resources based on their keys. A network that supports such queries is referred to as Content Addressable Network (CAN). Many solutions have been proposed to building CANs.

However most of these solutions do not focus on adversarial faults, which might be critical to building a censorship resistant peer-to-peer system. In a recent paper Fiat and Saia have proposed a solution to building such a system.

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 hypercube. We also demonstrate how this network can be maintained dynamically. This addresses the first open problem in the paper by Fiat and Saia.

*Department of Computer Science, Stanford University, Stanford, CA 94305. Email:datar@cs.stanford.edu.

1 Introduction

Peer-to-peer systems and applications are distributed systems without (ideally) any centralized control or hierarchical organization, which make it possible to share various resources like music [6, 2], storage [4] etc over the internet. Some of these systems have become very popular and have spurred recent interest in the research community. In order to share resources and access them over large, dynamic networks, users require means to locating them in an efficient manner. It is desired that given a unique key to a resource (like a URL, filename etc.) one should be able to access the resource over the distributed network. Thus what is desired is an efficient implementation of a hash table over such a dynamic and decentralized distributed system. Such a system is referred to as Content Addressable Network(henceforth CAN). Recently various solutions have been proposed for this problem [7, 8, 10]. While all of these solutions (architectures) are fairly robust against random attacks, a state or corporate agent that wishes to attack (make most or all of the content in the network inaccessible to majority of the users of the network) this system for political or commercial reasons, can do so by attacking carefully chosen points or nodes in the systems. For example, a centralized file sharing system like Napster [6] has been effectively dismembered by legal attacks on the central server. Additionally, the Gnutella [2] file sharing system, while specifically designed to avoid the vulnerability of a central server, has been studied [9] to be highly vulnerable to attack by removing a very small number of carefully chosen nodes. Recent work by Fiat and Saia [1] presents a content addressable network with n nodes that is censorship resistant, i.e. fault tolerant to an adversary deleting up to a constant fraction[1] of the nodes. There are more advantages to having such a system, beyond the main purpose of making it censorship resistant. While the previous solutions [7, 8, 10] discuss how their systems are fault tolerant to a few random faults, a detailed study of fault tolerance is lacking and not all of them provide guarantees about availability, low search time etc in the presence of a constant fraction of nodes being faulty. Most practical systems are vulnerable to such situations since the personal computers that participate in such systems may crash or lose network connectivity. Thus it is very desirable to have a system robust to a constant fraction of node deletes, a feature naturally present in censorship resistant system like the one in [1]. 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. Infact the first open problem that they mention in their paper (Section 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 solution is based on multi-butterfly networks. 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. In what follows we will first present a brief study of the previous solutions before describing our new solution.

Paper Organization: In Section 2 we will discuss the desirable features of a CAN and look at some of the metrics used in evaluating the goodness of such systems. In Section 3 we briefly study some of the recent solutions. In Section 4 we present a simpler and better censorship resistant network. Section 5 provides a dynamic construction of our network. Finally we conclude with a discussion of open problems in Section 6.

[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.

2 Desiderata and Metrics

Most CANs are built as an overlay network over the underlying network of links and routers connecting the different nodes in the system. A directed edge from a to b in an overlay network represents that node a has the necessary information (ip-address, port number etc.) to communicate with b whenever desired. Resources (Data items) are distributed amongst the nodes to be *maintained* by them. A single data item may be maintained by multiple nodes. Querying for any data item based on its key, typically involves routing the query through different nodes, using the edges in the overlay network, to reach a node that maintains that particular data item. At any given time we have a certain number n of nodes *participating* in the network. Of these there may exist nodes which may be unreachable because of some system failure, loss in network connectivity or because they are sabotaged by an adversary and are termed as *dead* or *faulty*. The remainder are termed as *live*. All solutions assume that a node that wishes to *join* has access to at least one *live* node in the network. Here are some of the desirable features and metrics that are used to evaluate the goodness of any proposed solution.

- **Decentralization:** The system should (ideally) be fully distributed and not have any hierarchy where some nodes are more important than others.
- **Load Balance:** In order to reduce the storage and lookup load for every node it is desired that each node should *maintain* an average (or close to average) number of data items per node.
- **Lookup Cost:** The cost of any query (lookup) is the sum of the weights of edges on the path used for the query. The weight of an edge in an overlay network is the network latency for a connection between the two nodes. This reflects the latency that a user sees when she issues a query. We assume that network latency is the dominant cost. It is one of the most important metric. However most of the proposed solutions (including ours) do not consider the weighted case and instead consider all edge lengths to be one. Thus they only aim to minimize the number of hops. It is desired that the lookup cost (number of hops) grows $O(poly(\log n))$ in order that the system is scalable.
- **Cost of join and leave:** Since such systems are expected to be highly dynamic in nature the system should adapt efficiently as nodes leave and join the network.
- **Degree in the overlay network:** Every node should have minimal degree in the overlay network. The degree of a node is directly proportional to its memory requirement and the ambient network traffic (like pings) that it may generate.
- **Fault/Censorship Tolerant:** The system performance should not degrade (provably) if some nodes become faulty or unreachable. Moreover most or all of the content in the network should be still accessible. A yet better desired feature, which has been justified earlier, is censorship resistance.

Besides these there may be some other desirable features which are particular to certain applications, like anonymity in search, anonymous posting of data etc.

Next we will take a look at some of the recent proposed solutions for CANs. Towards the end we will present a chart that gives asymptotic numbers for some of the important metrics discussed above comparing different solutions.

3 Related Work

In this section we will review some of the recent solutions [10, 8, 7] to building CANs. A common feature to all of these solutions [10, 8, 7] is the use of an underlying hash space to which nodes and data items are hashed deterministically and uniformly. The size of the hash space is large enough to

avoid collusions. Nodes are hashed using ip-address and data items are hashed using their key. It is important that data items are hashed deterministically and that every node can compute this hash value given the key of the data item.

The *CAN* system designed by Ratnasamy, Francis et al [8] uses a virtual d-dimensional (for a fixed d) Cartesian coordinate space on a d-torus. Every node has degree $O(1)$ and routing path length is $O(dn^{1/d})$.

The hash space used by *Chord* [10] and *Viceroy* [7] 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. The degree of every node is $O(\log n)$ and routing requires $O(\log n)$ hops whp. The *Viceroy* solution improves upon the *Chord* solution in that every node has a constant degree. It tries to maintain a butterfly network in a dynamic and decentralized manner. Similar to *Chord* routing takes $O(\log n)$ hops whp.

None of the above solutions focussed on adversarial node deletes. As mentioned in the introduction, if we desire to make the system censorship resistant then it is necessary to have the system fault tolerant to adversarial node deletes. Fiat and Saia [1] have developed a censorship resistant network (henceforth CRN). CRN departs from the previous work in that it is not dynamic and instead assumes that there are n nodes participating in the network, where n is fixed. The focus is instead on making the network robust to deletion of up to $n/2$ nodes by an adversary. The aim is to build a network such that even after an adversary deletes (makes faulty) $n/2$ 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.

They construct a network where the degree of every node is $O(\log n)$. Routing takes $O(\log n)$ hops and $O(\log^2 n)$ messages are sent for each query. Moreover the load on every node is $O(\log n)$ times the average load. Their network has the desirable censorship resistance, i.e. even after an adversary deletes a constant fraction of the nodes, all but ϵn of the remaining nodes have access to all but ϵ fraction of the data items. Moreover their network can be made resistant to spam by increasing the connectivity of every node to $O(\log^2 n)$ and increasing the number of messages sent for each query to $O(\log^3 n)$.

4 Multi-Butterfly Network (Multi-Hypercube)

In this section we present a censorship resistant network based on multi-butterfly networks, 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. The expected number of data items maintained by each node in CRN is $O(\log n)$ times the average, while in our network it is $O(1)$ times the average.

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 and does not change. Later on we will provide a dynamic construction that maintains an *approximate version* of this network that has similar properties.

Our construction is based on multi-butterflies. Figure 3 shows a twin-butterfly. Multi-butterfly networks were introduced by Upfal [11] for efficient routing of permutations and were later studied by Leighton and Maggs [5] for their fault tolerance. Please refer to their papers for details on multi-butterflies.

Butterflies and Multi-butterflies belong to class of networks that are often referred to as splitter networks (please refer to Figure 2). The basic building block of a splitter network is a splitter(please

[2]While the *Chord* paper [10] describes their hash space as "identifier circle modulo 2^m" the two are equivalent.

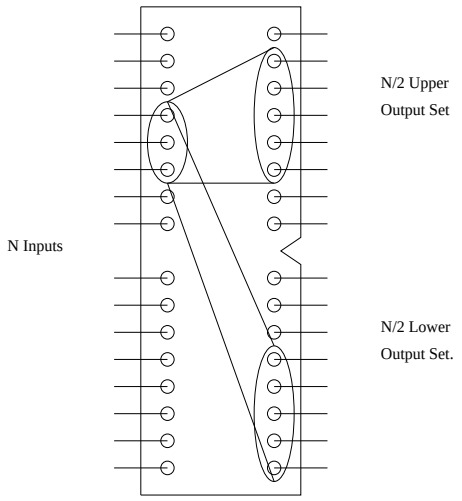

Figure 1: Splitter with N inputs and N outputs.

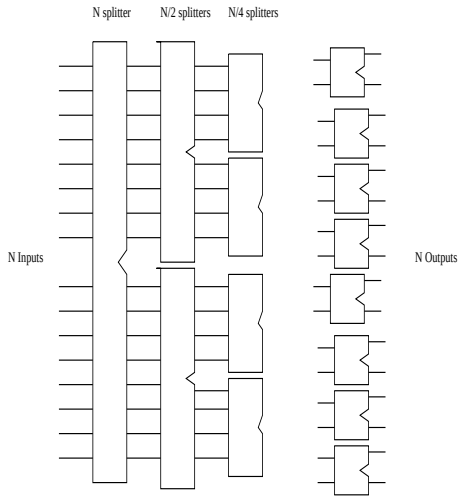

Figure 2: A splitter network with n rows, $\log n + 1$ levels.

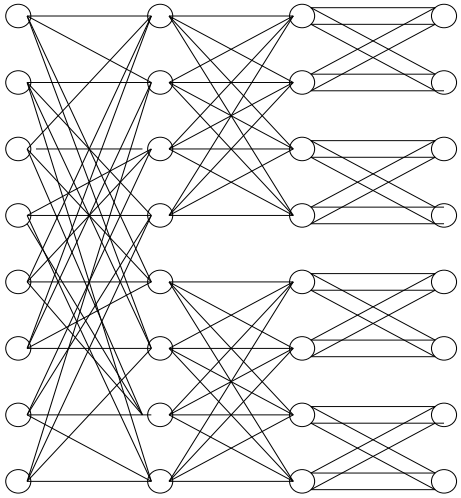

Figure 3: Twin Butterfly with 8 inputs.

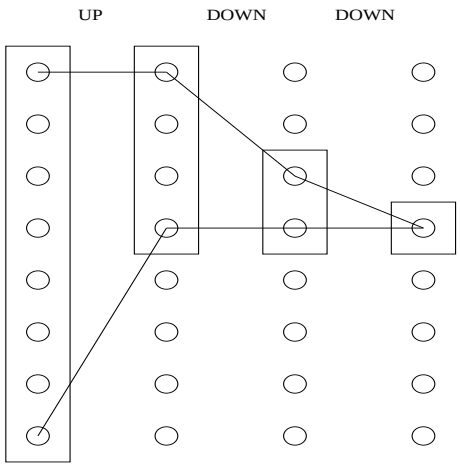

Figure 4: Logical path to an output node.

refer to Figure 1). An N-input splitter is said to have (α, β)-expansion if every set of $k \leq \alpha N$ input nodes is connected to at least βk up output nodes and βk down output nodes, where $\alpha > 0$ and $\beta > 1$ are fixed constants (Refer to Figure 1). Thus, the input nodes and the upper (lower) output nodes form a concentrator from N nodes to $N/2$ nodes with (α, β)-expansion. A multi-butterfly is said to have (α, β)-expansion if all its splitters have (α, β)-expansion. Splitters with expansion are known to exist for any $d \geq 3$, and they can be constructed deterministically in polynomial time [11], but randomized wirings will typically provide the best possible expansion. Infact 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 [11]).

In a splitter network, each input and output are connected by a logical (up-down) path through blocks in the network. Please refer to Figure 4. In a simple butterfly since there is only one up edge and down edge this logical path corresponds to a unique path through the nodes in the network. However this is not the case in multi-butterflies with multiplicity d, where we have a choice of d edges at each node. Hence one logical path can be realized by a myriad of paths in the network. It is this availability

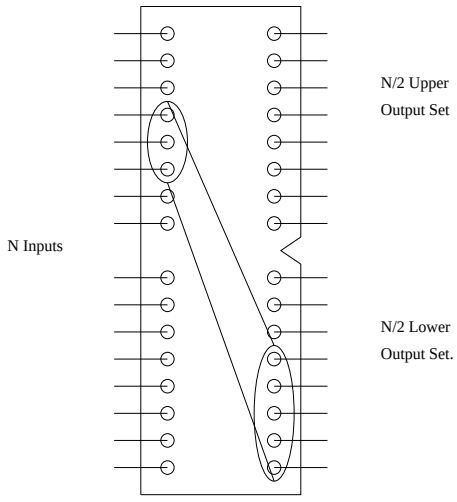

Figure 5: Splitter in a multi-hypercube.

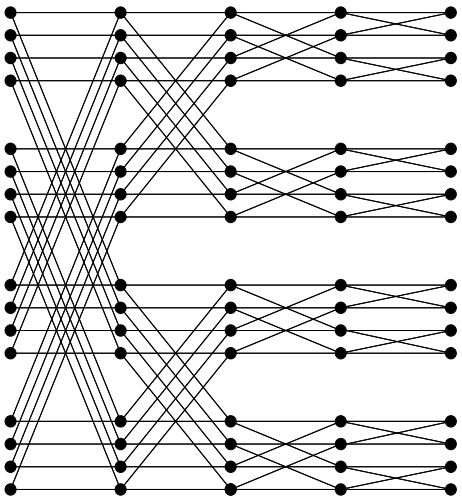

Figure 6: Butterfly network of 16 rows and 5 levels.

of choice that makes multi-butterflies much more robust than simple butterflies. A multi-butterfly is very robust at routing because one must block βk outputs in order to block k inputs. In a butterfly the reverse is true: it is possible to block $2k$ inputs by blocking k outputs. This effect when compounded over several levels, the effect is dramatic. In a butterfly, a single fault can block 2^l nodes l levels back, while in a multi-butterfly, it takes β^l faults to block a single node l levels back. This insight is the key to the fault tolerance of a multi-butterfly which was studied by Leighton and Maggs [5].

While the nodes in a multi-butterfly have constant degree, a multi-butterfly with $\Theta(n \log n)$ nodes can only tolerate $O(n)$ faults, in order that $\Omega(n)$ input nodes are still connected by a logical path to $\Omega(n)$ output nodes. As a result it is not suited for Censorship Resistance. Hence we build a new network called multi-hypercube that is based on multi-butterflies and is the fault tolerant version of hypercube. In short, a multi-hypercube is to a multi-butterfly in what a hypercube is to a butterfly. If the role of all the nodes in a single row of a butterfly is played by a single node then what we get is a multi-hypercube. The only caveat is that in every splitter we only maintain the "cross" edges and not the "straight" edges. In other words every input node in a splitter connects to d nodes from the upper output nodes or the lower output nodes, but not both, depending on its position. The single "straight" edge connecting every input node to the corresponding output node is obtained for free as the two nodes are same. The splitter in a multi-hypercube is shown in Figure 5. In this splitter, upper $N/2$ input nodes are connected to the lower $N/2$ output nodes and vice versa, with an expander of degree d. As a result we get better expansion factor (β) for the same degree as opposed to that in a multi-butterfly, since we have an expander from $N/2$ nodes to $N/2$ nodes instead of a concentrator from N nodes to $N/2$ nodes. To the best of our knowledge this network has not been studied earlier and neither are we aware of the use of the term multi-hypercube. A multi-hypercube can be defined as follows: 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 random nodes whose first i bits are $b_1 b_2 \ldots b_{i-1}\overline{b_i}$. It also has "in-edges" from d random nodes whose first i bits are $b_1 b_2 \ldots b_{i-1}\overline{b_i}$. The expansion property holds at each level, like in the case of a multi-butterfly. Thus a multi-hypercube with n nodes has degree $2d \log n$ for each node. It turns out that multi-hypercube is ideal for censorship resistance.

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 butterfly. The data items are randomly hashed onto any node. As a result every node maintains average number of data items in an expected sense. Using consistent hashing as in [10] 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 the number of smaller identifiers. Thus at the end of round one every node knows its index i in the sorted list. Based on i every node knows the indexes of all other nodes that it will connect to at different levels. Note that every node connects to at most $2d \log n$ other nodes. In round two every node will broadcast its index i and its identifier (ip-address) to all other nodes. Every node in turn will remember the identifiers of the pertinent $2d \log n$ nodes that it should connect to. It will then form a connection with these. As mentioned before data items are randomly hashed to 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 $2n$ broadcasts with $2n^2$ messages and assumes that each node has $O(\log n)$ memory, similar to the creation of CRN.

Routing: Routing between any pair of nodes is done in the obvious way as in a hypercube using the logical or bit-correcting path. Every node that wishes to access a data item computes the hash of its key and finds out the index of the node it belongs to (destination). The query is then routed between this node and destination. Note that, for a fault free multi-hypercube, at each level (each splitter) we have a choice of d out edges. We can take any *one* of them that connects to a live node. However, as we will see, if some of the nodes in the multi-hypercube become faulty then our choice of out edges may reduce at each level. The number of messages sent for a single query is at most $\log n$ and time taken is also $\log n$. Each node has indegree and outdegree exactly $2d \log n$.

Next we prove that this network is fault tolerant. The proof is similar to that presented in [5]. The difference is that while every node is distinct in their case, in our case every node plays the role of $1 + \log n$ nodes in the same row.

4.1 Fault Tolerance

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 an input node and output node, for that matter all the nodes in a row. We will prove that no matter how an adversary selects f nodes to be faulty (deletes them), there are always at least $n - O(f)$ inputs and $n - O(f)$ outputs such that between any pair of them there still exists a logical path of length $\log n$ such that all nodes on the path are not faulty and can be used for routing as described above.

We first describe which outputs to remove. Note that this "removal" is only logical for the purpose of counting and the nodes are not removed in practice. 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. 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 $\frac{f}{\epsilon} = \frac{f}{\alpha(\beta-1)}$ outputs are removed by this process.

We next describe which inputs to remove. Working from the $\log n$th level backwards, examine each

node to see if all of its upper (or lower depending on its position in the splitter) outputs lead to faulty nodes that have not been erased. Note that every node is connected to d nodes from either upper or lower outputs. 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 an $\alpha/2$ fraction are propagated by faulty upper outputs and at most an $\alpha/2$ fraction are propagated 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 $\alpha/2$ fraction of the propagated faults from its upper outputs, then more than $\alpha m/2$ 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 $\frac{\epsilon_O m}{2}$ faults), however, we know that $|U| < \frac{\alpha m}{2} + \frac{\epsilon_O m}{2} = \alpha\beta m/2$ which is a contradiction. Hence at most $\alpha/2$ fraction of the inputs of any splitter are propagated from any faulty upper(or 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. By previous Lemma, we know that these faults are divided among splitters linking level l to level $l+1$ so that we can apply the expansion property to the faults within 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 the output nodes that are not erased. This leaves a network with $n - \frac{\beta f}{\beta-1}$ input nodes and $n - \frac{f}{2\alpha(\beta-1)}$ outputs nodes such that every remaining input has a logical path to every remaining output. While the algorithm above gives an off-line 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 [3] 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 allows us to label nodes "faulty" so that we are careful not to forward a query to such faulty nodes. Note that this restricts the choice of edges along which we can forward the query as the network becomes faulty. We suggest that the network does this reconfiguration in a periodic manner.

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 $n - \frac{f}{2\alpha(\beta-1)}$ nodes.*

We can choose the multiplicity d and parameters α, β such that $\alpha(\beta-1) \geq 2/3$ and $\beta \geq 3$. We will construct our network where every splitter has the parameters above. Substituting these values in the theorem above we get that no matter which f nodes are made faulty, there are at least $n - \frac{3f}{2}$ nodes that can reach $n - \frac{3f}{2}$ node through a logical path. Note that the guarantee above is deterministic as opposed to whp as in case of CRN. 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. It guarantees that as long as the faults are less than $n/2$, whp a constant fraction of top supernodes can reach a constant fraction of bottom supernodes. It does not characterize the behavior when the number of faults are sublinear.

If the number of faults is $n/2$ then we have that $n/4$ nodes can reach $n/4$ nodes.

We can guarantee that $(1-\epsilon)$ fraction of the data is available to $(1-\epsilon)$ fraction of the remaining live nodes (whp) by doing the following two things similar to that in CRN: Instead of hashing the data items to just one node we can hash them to $k_1(\epsilon)$ (a constant that depends on ϵ) nodes. Also in addition to the network that we have, we can have all nodes connect to $k_2(\epsilon)$ random nodes on the "top", through which they can route the queries if they themselves do not have path to nodes at "bottom". That way we are guaranteed that whp $(1-\epsilon)$ fraction of the remaining live nodes have access to large fraction $(1-\epsilon)$ of the data items. Note that this last extension has results that are not deterministic.

Also note that we can change the network above to have $n/\log n$ rows and $\log n - \log\log n + 1$ levels, effectively maintaining a multi-butterfly with $n/\log n$ rows. In that case every node participates in only one level. This way we can ensure that nodes have constant degree instead of logarithmic. The problem with this approach is that such a network cannot tolerate linear number of faults. It can only tolerate $O(n/\log n)$ faults. But for some cases that may be a better choice, particularly if constant degree feature is more attractive than being able to withstand linear number of faults.

The network described above has better fault tolerance for random faults. However we are not concerned with that aspect since the focus is on adversarial faults.

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 $d\log n$.*
- *Every data item is randomly and uniformly hashed onto one of the nodes. As a result the expected data items stored at every node is equal to the average.*
- *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.*
- *The network above can be enhanced by hashing data items onto multiple, although constant ($k_1(\epsilon)$), nodes and having each node connect to $k_2(\epsilon)$ other random nodes so that $(1-\epsilon)$ fraction of the live nodes can access $(1-\epsilon)$ fraction of the data items as long as no more than $n/2$ nodes are faulty.*

5 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 time the network that is built is fault tolerant to adversarial faults, we cannot add more nodes to the network once adversarial faults happen. 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* [10] and *Viceroy* [7] 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 collusions. $Successor(x)$ 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*. Similar to *Chord* we maintain a successor list of size $r = O(\log n)$ so that our network is fault tolerant to random faults. 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.

The main idea of our construction is that every node maintains outgoing edges with d random nodes that belong to a dyadic interval at certain level (of certain width). It does so for $O(\log n)$ levels. What follows are details about how to do this in dynamic manner as nodes join and leave and taking care of imperfections that may occur due to random hashing.

5.1 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 interval other than the one it belongs to. 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 $1 - 1/n^2$, $max(\frac{k_u}{k_l}, \frac{k_l}{k_u}) < 1/2$.*

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. As a result if k is the total number of nodes that belong to the interval pair, the expected number of nodes that belong to each interval is $k/2$. 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.

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 [7](Lemma 4.3)

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 $\frac{n-1}{2\log n} \leq n_o \leq n^3$.*

The lemma says that while a node may not estimate n too well it can estimate $\log n$ within constant factor. Infact it follows that whp $\log n \leq 2\log n_0 \leq 6\log n$. In our construction every node will use this estimate of $\log n$. A node may be over estimating this quantity by a constant factor, but that does not affect our results.

5.2 Dynamic construction.

A query in our network is of the form $successor(x)$ where given a value x we return the node $successor(x)$. This is useful to find a data item with identifier x and also during the construction of the network to find the appropriate nodes that a given node will connect to. We assume that we can answer this query while we describe the dynamic construction.

Node Join: A node that joins the network has access to some live node in the network. It will use this node to issue 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. It also maintains a successor list of size $r = O(\log n)$ similar to *Chord* which makes the network resilient to random faults.
- 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 pair.
  - If there are less than $c\log n$ nodes in $DI(x,i)$ maintain edges to all of them and set $done = true$.
  - Else choose d random values from the interval $DI(x,i)$. Let $r_i^1, r_2^i, \ldots, r_d^i$ be the set of random values chosen from the interval. For every value z in this set issue the query $successor(z)$ and maintain an edge with this node.
  - increment i.

  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 these nodes. 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 now describe in short how these edges are maintained over time as nodes leave and join. For every value r_k^i that the node has randomly chosen from the dyadic interval $DI(x,i)$, the node periodically issues successor queries and checks if the node that it maintains an edge with is indeed $successor(r_k^i)$. This is necessary because as nodes join and leave $successor(r_k^i)$ may change. However the nodes need not do this too often. It is sufficient to do this as the number of nodes reduce or grow by a factor 2, i.e. whenever their estimate of $\log n$ changes. We can also follow a more proactive protocol: When a node leaves the network it informs the nodes that maintain an edge with itself. These nodes then connect to the successor of the departing node. A node also periodically

checks to make sure that the smallest dyadic interval that it maintains edges to has no more than $c \log n$ nodes. If the number of nodes exceed $c \log n$ it further "splits" this interval using the procedure described in join. Similarly it may "merge" the smallest intervals if too many nodes leave the network. **Node Leave:** Similar to *Chord* a node that wants to leave transfers its data items to its successor. It may inform the nodes that have an incoming edge to it if we follow the more proactive protocol. Alternatively we may rely on the network to correct itself periodically as described above.

5.3 Routing:

Consider a query $successor(y)$ starting at node x. Let $y = 0.y_1y_2y_3 \dots y_p$ be the binary representation of y. We consider the starting node x to be at level 0. The node looks at the first bit of y (y_1). If y_1 differs from x_1 it forwards the query to any node from the dyadic interval from $DI(x, 0)$. In doing so it has the choice to forward it any of the d random nodes that it maintains connections to from this interval. If y_1 agrees with x_1 then the node does nothing and the query enters the next level. The query is now in level 1. In general consider a node z that receives this query at level i. Its easy to see that its identifier matches with that of y in the first i bits. Based on the $i+1$th bit y_{i+1}, if it differs from z_{i+1}, it forwards it to the dyadic interval $DI(z, i)$ or else it does nothing. Again as before it has a choice to forward it to any of the d nodes. Finally when the query reaches a node w at level l such that at this level the node maintains edges with all the nodes in the dyadic interval it finds $successor(y)$ and forwards it to $successor(y)$. Based on the query, $successor(y)$ returns the appropriate data item or information about itself. The answer is returned along the path that the query was forwarded along. This the bit correcting logical path in the multi-hypercube. Search takes $O(\log n)$ hops and $O(\log n)$ messages are sent.

5.4 Fault Tolerance:

In our construction we maintain that every node x maintains outgoing edges with d random nodes from every dyadic interval pair at different levels. If the number of nodes in the interval drops below $c \log n$ the node maintains edges with all of them. Moreover from Lemma 3 we have that dyadic interval pairs with more than $c \log n$ nodes are balanced whp. It follows from Lemma 4.1 in [1] that whp (at least $1 - 1/n^2$), such intervals (with atleast $c \log n$ nodes) will have the crucial expansion property for parameters α, β that satisfy $2\alpha\beta < 1$. The factor 2 comes from the slight imbalance that the interval pair may have in the number of nodes. Thus we maintain the crucial expansion property that is required for fault tolerance, in a randomized manner. In MBN we achieved this in a deterministic manner by using explicit expanders.

The proofs for fault tolerance follow exactly as in MBN. We replace splitters with the dyadic intervals and the arguments follow. The only small point is that in MBN the splitters were exactly balanced, where we used the expansion property. In case of DMBN we know that they may be slightly unbalanced, but only by factor 2. This can be taken care by defining $\epsilon_0 = \alpha(\frac{\beta}{2} - 1)$ in the counting argument that deleted the output and input nodes. Note that our guarantees are no more deterministic since we are maintaining the splitter connections in a randomized manner. We get the theorem

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.*

Again as before we can increase the data availability to large portion of the remaining nodes by hashing the data items multiple times and having nodes connect to constant number of other nodes from where they issue queries.

It is important to note why the dynamic construction wont work after an adversarial attack has taken place. The construction assumes that all $successor(x)$ queries will be answered correctly. This is

Network	linkage (degree)	query cost (path length)	Messages per query	Fault Tolerance	Dynamic	Data Load (times average)
CAN [8]	$O(1)$	$O(n^{1/d})$	$O(n^{1/d})$	?	Yes	$O(1)$
Chord [10]	$O(\log n)$	$O(\log n)$	$O(\log n)$	Random	Yes	$O(1)$
Viceroy [7]	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	$O(1)$

Figure 7: Comparison of recent solutions.

necessary part of the construction, for new nodes to establish their edges. However once an adversarial attack has taken place such a guarantee cannot be given. Infact it is easy to see that an adversary may delete all nodes on some continuous segment of the unit circle and successor queries in that segment will fail. 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.

6 Conclusion and Open Issues

We present a new network called multi-hypercube, a fault tolerant version of hypercube, that is ideal for building censorship resistant networks and improves upon the previous solution by Fiat and Saia [1]. We refer to this network as MBN. We also present a dynamic version of MBN called DMBN that has similar properties to MBN. In doing so we address the first open problem in [1]. We now present a table (refer to Figure 7) that compares all of these solutions based on some important factors:

We mention here some of the open issues that remain to be addressed for fault tolerant CANs:

- Can we build an "efficient" dynamic CAN that is fault tolerant to adversarial faults and allows dynamic maintenance even after an adversary deletes a constant fraction of the existing nodes.
- 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 of a network that is fault tolerant to linear number of adversarial faults.

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, San Francisco, CA, USA. Jan. 2002.*

[2] Gnutella website. `http://gnutella.wego.com/`.

[3] A. Goldberg, B. Maggs, and S. Plotkin. A parallel algorithm for reconfiguring a multi-butterfly network with faulty switches. In em IEEE Transactions on Computers, 43(3), pp. 321-326, March 1994.

[4] 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 em Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS 2000), Boston, MA, USA, November 2000.

[5] 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.*

[6] Napster website. `http://www.napster.com/`.

[7] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Lookup Network. Submitted to *Thirty-Fourth Annual ACM Symposium on Theory of Computing, 2002.*

[8] 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.*

[9] 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.

[10] 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, San Diego, CA, USA, August 2001.*

[11] E. Upfal. An $O(\log N)$ deterministic packet-routing scheme. In *Journal of ACM, Vol. 39, No. 1, Jan. 1992, pp 55-70.*