Distributed Construction of a Fault-Tolerant Network from a Tree

Michael K. Reiter[1] Asad Samar[2] Chenxi Wang[2]

February 2005
CMU-CS-05-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1] Department of Electrical & Computer Engineering and Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; reiter@cmu.edu

[2] Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; {asamar, chenxi}@ece.emu.edu

Keywords: fault-tolerant networks, expander graphs, random walks, self-stabilization, distributed algorithms

Abstract

We present an algorithm by which nodes arranged in a tree, with each node initially knowing only its parent and children, can construct a fault-tolerant communication structure (an expander graph) among themselves in a distributed and scalable way. Tree structures arise naturally in many distributed applications, in which a node "joins" the system by contacting a node already present in the system: the joining node then becomes a child of the node it contacts for entry. Our algorithm enables nodes to construct an expander graph incrementally without propagating membership information globally. At the core of our construction is a novel distributed mechanism that samples nodes uniformly at random from the tree. In the event of node joins, node departures or crash failures, our graph self-stabilizes to a state where it still achieves the required fault tolerant properties. We present simulation results to quantify the convergence of our algorithm to a fault tolerant network having both good vertex connectivity and expansion properties.

1 Introduction

In this paper we consider the problem of constructing a fault-tolerant communication network—i.e., one having redundant paths between any pair of nodes—among nodes that are initialized in a tree topology. Tree structures are commonly seen in network applications because of their resemblance to organizational hierarchies (e.g., the DNS name space). Tree structures, however, exhibit poor fault tolerance properties, in that a single node or link failure can partition the tree. It is therefore prudent to build from this initial tree structure a logical communication network that has better ability to tolerate faults.

Here we present a distributed algorithm to do this by building an expander from the tree. Expanders are an important class of graphs that have found applications in the construction of error correcting codes [28], de-randomization [1], and in the design of fault tolerant switching networks [26]. The fault tolerant properties of expanders [13][4] are precisely what motivated this research. Our algorithm starts with nodes connected in a logical tree structure and proceeds to add edges to achieve an expander. However, because explicit constructions of expanders are generally very complex, we specifically present a construction that "approximates" a d-regular random graph, i.e., a random graph in which every node has almost *d* neighbors. A d-regular random graph is, with an overwhelming probability, a good expander [10].

The contributions of this work rest primarily in two features. First, our algorithm is completely distributed. Though expander graphs have been studied extensively, distributed construction of expander networks remains a challenging problem. Each node in the network keeps state for only its neighbors and the algorithm uses local information at each node. A direct consequence of this is scalability—our algorithm is capable of generating expanders efficiently even with a large node population. We bootstrap this algorithm using a novel technique that samples nodes uniformly at random from the tree regardless of the tree size—most previous attempts at distributed uniform sampling require the number of nodes to be sufficiently large and otherwise resort to centralized or broadcast-based approaches. This bootstrapping technique is also decentralized but requires nodes to propagate some information across the tree. We show that this can be done with low message complexity.

Second, our algorithm adapts to dynamic node joins and leaves, converging back to an expander after such a perturbation. Previous attempts at distributed construction of random expanders [17][12] try to construct d-regular random graphs where every node has exactly *d* neighbors. Such graphs are difficult to construct and maintain in a dynamic distributed setting. We follow a more pragmatic approach, in that we only require that nodes have "close" to *d* neighbors. This gives us more flexibility in dealing with the dynamic nature of our network, while still achieving the fault tolerant properties. One consequence of using this approach is that we do not require nodes to notify others when leaving the network, thereby accommodating crash failures. To the best of our knowledge, distributed construction of expander networks that self-stabilize even when a large fraction of nodes crash, is an open problem that has not been addressed previously.

While our algorithm has application in any scenario in which building a fault-tolerant network from an initially minimally connected system is warranted, we arrived at this problem in an effort to enhance the fault tolerance of a particular mechanism. This mechanism, called *capture protection* [23], protects a cryptographic key from being misused even if an adversary captures and reverse engineers the device (e.g., laptop) on which the key resides. Capture protection achieves this property by requiring the consent of a remote *capture protection server* in order for the cryptographic key to be used, which the server will give only if it can authenticate the current user of the device as the proper owner. This server, however, does not need to be fixed; rather, one server can *delegate* the authority to perform this role to a new server [22], giving rise to a tree structure in which the new server is a child of the server that delegated to it [27]. The need for a more fault-tolerant structure arises from the need to *disable* the device globally when its capture is discovered, to eliminate any risk that the adversary finds a way to impersonate the authorized user to a server. As we would like this disable operation to succeed globally despite server failures (possibly attacker-induced) in the tree of authorized servers, we approached the problem that we address in the present paper.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces some background material. Section 4 describes our system model and briefly outlines the goals of this work. Section 5 presents the expander construction algorithm and related proofs. Section 6 describes simulation results. We conclude in Section 7.

2 Related work

Expander graphs are a well studied design for fault-tolerant networks. Both randomized [29],[15] and explicit [24], [11] constructions of expanders have been known for sometime. However, little has been done to construct expander networks in a distributed setting.

Law and Siu [17] presented a distributed construction of expander graphs based on 2d-regular graphs composed of *d* hamiltonian cycles. However, to sustain expansion properties of the graph in the event of nodes leaving the system, they require that a leaving node send its state to some other node. This approach cannot tolerate crash failures. Furthermore, their approach requires obtaining global locks on the hamiltonian cycles when new nodes join, which can be impractical in a large distributed system. Finally, they revert to employing either a centralized approach or using broadcast when the number of nodes is small since their mechanism can only sample uniformly at random from a sufficiently large number of nodes.

Gkantsidis, Mihail and Saberi [12] extend the mechanisms presented by Law and Siu [17] to construct expanders more efficiently, i.e., with constant overhead. However, their approach uses *d* processes in a *2d*-regular graph, called "daemons". These daemons move around in the topology. Every joining node must be able to find and query such a process. Thus their system is not completely decentralized.

Pandurangan, Raghavan and Upfal [25] present a distributed solution to constructing constant degree low diameter peer-to-peer networks that share many properties with the graphs we construct here. However, their proposal also employs a centralized server, known to all nodes in the system, that helps nodes pick random neighbors.

Loguinov et al. [19] present a distributed construction of fault resilient networks based on de Bruijn graphs that achieve good expansion properties. However, they also require nodes leaving the system to contact and transfer state to existing nodes and thus cannot tolerate crash failures.

3 Background material

In this section we present some known results from the theory of random regular graphs and random walks. These concepts are used in the subsequent sections.

3.1 Random regular graphs

Let 5(n, *d)* denote the set of all d-regular graphs on n nodes and G_n*,d* be a graph sampled from *S(n, d)* uniformly at random. Then $G_{n?}$d is a *random regular graph.* It is known that random regular graphs have asymptotically optimal expansion properties with high probability [10].

Configuration model [6] is the standard method for generating random d-regular graphs on *n* nodes *v*, ^2,..., v_n, though not in a distributed setting. In this model each vertex is represented as a set containing *d* points, resulting in *n* such sets, 7(^1), 7(^2), ...,7(v_n)- A *perfect matching* of these *nd* points is a set of *\nd* pairs of points such that no two pairs share a common point. Assuming *nd* is even, many perfect matchings exist for these points. A *uniform random perfect matching* is a perfect matching chosen uniformly at random from the set of all possible perfect matchings. To construct a random d-regular graph on n vertices, a uniform random perfect matching on these *nd* points is computed and an edge is inserted in the graph between vertices *Vi* and *Vj* if and only if the perfect matching pairs a point in *j(vi)* to a point in *J{VJ).* This model allows self loops and parallel edges and is very inefficient if the goal is to construct a *simple graph*, i.e., one without self loops and parallel edges. A refinement [29] of this model constructs random d-regular simple graphs by pairing points, one pair at a time, from the uniform distribution over all available pairs, i.e., those that do not result in self loops and parallel edges. Graphs generated using this approach are asymptotically uniform for any $d \leq \mathrm{n}^{1}/^{3''6}$, for any positive constant e [15].

3.2 Uniform sampling using random walks

A random walk on a graph can be modeled as a Markov chain. For a graph containing *n* nodes, the *probability transition matrix M* of the random walk is an n x *n* matrix where each element *Mij* specifies the probability with which the random walk moves from node *i* to node *j*. Let *nt* be a vector such that *nt[i]* is the probability with which the random walk visits vertex *i* at step *t*. Then 7Tt+i = $n_t M$ = $7\mathrm{ToM}^{t+1}$. A vector 7T is called the *stationary distribution* of the random walk if *n* = *nM*, i.e., the stationary distribution remains the same after the random walk takes a step, or any number of steps for that matter. It is known that a random walk on a connected undirected graph with an odd cycle has a unique stationary distribution [20]. *Mixing time* is the time required for the random walk to reach its stationary distribution and it depends on the expansion properties of the graph: the walk reaches stationary distribution quickly if the graph is a good expander. A random walk on a graph can be used to sample nodes from the walk's stationary distribution if the walk is run long enough to mix properly.

Let *TG(X)* denote the set of neighbors of node *x* in graph *G* (we define this formally later). Then a *simple random walk* is a walk which, at each step, moves from a node *x* in *G* to one of its neighbors in *TG(X)* with probability *l/\rc(x)*. The stationary distribution of a simple random walk on a regular graph is uniform, i.e, TT = ^[1,1,..., 1] . In case the graph is not regular, the stationary distribution of a simple random walk is a function of the nodes' degrees. One of the known ways (recently also discussed in [3],[7]) to sample uniformly at random from an irregular graph *G* with maximum degree dmax is to run a random walk on *G* that takes a step from node *x* to node *y* with probability:

$$P_{xy} = \begin{cases} \frac{1}{d_{\max}} & \text{if } y \;\hat{}\; x \text{ and } y \text{ G } TG(X) \\ 1 - \frac{|\Gamma_G(x)|}{d_{\max}} & ify = x \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

We call such a random walk a *maximum degree random walk* and denote it as MDwalk. An MDwalk has a uniform stationary distribution even on irregular graphs but it suffers from two main issues: (i) In a dynamic distributed system it might be difficult to estimate the maximum degree of the graph, (ii) Low degree nodes imply higher self transition probabilities (see Equation 1) which result in longer mixing times for MDwalks. If MDwalks are not run long enough to achieve sufficient mixing, they are biased towards low-degree nodes.

4 System model and goals

Our system consists of a set of nodes distributed over a network that is structured as a rooted tree and denoted as *T* = (V, *ET)* (vertex set remains the same across the different graphs discussed here but the edge sets differ, hence the subscript). For any subset S c K w e define the set of *neighbors* of *S* in *T* as *FT(S)* = *{y* G *V* | *3x* G 5, *(x,y)* G J5r}. Nodes are initialized only with the identities of their neighbors and do not have access to any central database containing information about T.

Nodes are allowed to join and leave the tree. We further allow nodes to crash. A crashed node may recover and re-join the tree at a later time. Crash failures are modeled as nodes leaving the tree, so nodes are not required to send any special messages when leaving.

We work in an asynchronous model, i.e., messages may take arbitrarily long to reach destinations and nodes do not have a common clock. We assume the existence of an *unreliable failure detector* [9] at each node *x* that is used only to detect crashes among the neighbors of *x* in the tree; detecting failures elsewhere in the tree is not required. We assume that during the execution, there are "periods of stability" when our network is static, the failure detector achieves strong completeness and most correct processes are not suspected (a very weak form of accuracy). Such an unreliable failure detector can be easily implemented in practice using a timeout mechanism.

We present some notation used to define expander graphs.

Definition 1. *Given a graph G =* (V, *EG), the vertex boundary* d_G{S) *ofa set S* C *V is* d_G(S) = {yG *V\S\3xeS,* *(x,y)e*E_G}.

Definition 2. *A graph G = {V,*E_G*) is an a-expander if for every subset S* C *V of size \S_< \V\/2,* \9G(S)\ >_*a\S\, for some constant a* > 0.

Our goals can be summarized as follows: Construct an expander graph with the same vertex set as *T* using a distributed algorithm that scales well. New nodes should be able to join the expander quickly without incurring a high messaging cost and existing nodes should not be required to send special messages when leaving. Finally, the network should self-stabilize even after a large fraction of nodes crash.

5 Expander construction

Our approach is to construct a random graph among vertices in *V* (nodes in our network) such that nodes in the graph have degrees close to some constant *d.* Such a graph is much easier to construct and maintain in a dynamic distributed system than a d-regular random graph while still achieving good expansion.

5.1 Random almost-regular graphs

We say a graph is (d, *e)-regular* if the degrees of all nodes in the graph are in the range *[d* — e, *d* + e]. Let *S(n,* d, e) be the set containing all (d, e)-regular graphs on *n* nodes, then sampling uniformly at random from *S(n,d,e)* results in a (d, e)-*regular random graph,* denoted G_n^e- Note that one way to construct G_n,d,c is ^{t0} fi^ construct G_n^-$_t$, i.e., a *(d — e)*-regular random graph, and then add some random edges such that the degrees stay within *[d* — e, *d* + e]. Therefore, the expansion of G_n^,*t* is lower bounded by the expansion of G_n,d-$_O$ since adding edges to a graph does not decrease its expansion.

Our distributed construction generates (d, e)-regular random graphs by approximating the refinement [29] of the configuration model (see Section 3.1) as shown in Figure 1. Note that nodes continuously sample new neighbors from the tree but maintain a maximum of d + e neighbors.

A simple probabilistic argument shows that if all nodes sample and add neighbors with the same frequency, then the probability of a node having degree less than d — e decreases exponentially in e. Using e = d/2 gives a reasonably high confidence that all nodes will have degree greater than d — e even for small values of d like d = 10. Note that in case of crash failures, the degrees at some nodes may go below d - e, but the fault-tolerant properties of the expander will ensure that most nodes remain connected in a component that has high expansion. In this case, nodes with low degrees recover quickly when the network stabilizes.

Continuously sampling and refreshing expander neighbors (lines 2-5 of Figure 1) is needed for two reasons: 1) We expect the failure detector at node *x* to eventually detect the failure of nodes in *TT(X) (*X^9*S*

```
Every node x £ V executes the following:
Initialization:
1.  $T_G${X) <r- 0

Main:
2.  repeat forever
3.      uniformly sample node y from V
4.      send (Add : y) to x
5.      wait for a specified period

Upon receiving (Remove : y):
6.  remove y from TG{X)

Upon receiving (Add : y):
7.  if y = xory G $T_G(x)$
8.      do nothing
9.  else
10.     if $|\Gamma_G(x)| = d + \epsilon$
11.         pick z from F G ( # ) at random
12.         removes from F G ( ^ )
13.         send (Remove : x) to z
14.     add y to $T_G(x)$
15.     send (Add : x)toy
```

Figure 1: Algorithm to generate (d, e)-regular random graph

neighbors in the tree) but not of nodes in *TQ{X)* (X'S neighbors in the expander). It may be easier for the failure detector to detect crashes among neighbors in the tree because in most applications neighbors in the tree will be "close" to each other, either geographically or according to some network metric. Examples of such applications are recently proposed location aware peer-to-peer architectures [18],[8],[21]. Continuously refreshing the neighbor set *TG(X)* allows *x* to replace crashed nodes in this set without having to explicitly detect failures of possibly far-flung nodes. We delay further discussion on crash failures until Section 5.5. 2) Continuous sampling and refreshing allows us to ignore periods where we might be unknowingly sampling from a biased distribution (e.g., because the failure detector has not yet detected a crash) as these neighbors will later be replaced by nodes sampled from a distribution closer to uniform during stable periods.

These mechanisms reduce the problem of constructing G_n,d,cto that of a node *x* G *V* choosing another node uniformly at random from the tree (line 3 of Figure 1), i.e., with probability 1/|V|. A sampling procedure that picks a node in the tree uniformly at random could be used by *x* to construct and maintain *Gn,d,e* as described above.

5.2 Biased irreversible random walks

For a node, choosing another node uniformly at random from the tree is challenging due to the specific structure of the tree and the fact that each node only knows about its neighbors.

We approach this problem by assuming that every node *x* knows about the number of nodes in the tree in the direction of each of its neighbors (we relax this assumption in Section 5.3): For each child, *x* knows about the size of the subtree rooted at the child and for the parent, *x* knows about the number of nodes in the tree that are not in the subtree rooted at *x.* Then, to choose a node uniformly at random from the tree, *x* starts a biased irreversible random walk, Blwalk. At each step, the Blwalk either (i) moves from a node to one of its neighbors in the tree, except the neighbor where it came from or (ii) picks the current node. In case (i), the probability of choosing a neighbor is directly proportional to the number of nodes in the tree in the direction of that neighbor (we make this formal below). In case (ii), we say the Blwalk *terminates.* The node where the Blwalk terminates adds *x* to its neighbor set and notifies *x.* Upon receiving this notification, *x* also adds the sampled node to its neighbor set, thus forming an undirected edge. We prove that a Blwalk samples nodes uniformly at random from the tree during periods of stability when the tree is static and failure detectors satisfy strong completeness.

Let (x, *y)* be an edge in E_T (E_T is the edge set of T) and *F(V,* ET\{(X, *y)})* be the forest containing two components formed by removing (x, y) from T. Then, we define *C(x* —>• y) to be the component of *F* that contains node y. The '-V notation captures the intuition that this is x's view of the tree in the direction of its neighbor y. Let V^1 denote the vertex set of *C(x* -» y), then *W(x* -> y) = |V|. Intuitively, W(x -» y) represents z's view of the "weight" of the tree in the direction of its neighbor y, i.e., the number of nodes in the tree in the direction of y. For convenience, we define *W(x —> y)* = |V| if a: *& V* and y 6 *V* (the view from outside the tree), and *W(x* -» y) = 1 if a; = y (the view when a: looks down at itself).

We denote a BIwalk as a sequence of random variables *X\,Xi,* --,Y_9 where each Xj represents the node that initiates the i^{th} step of the BIwalk *(X* starts the BIwalk) before the BIwalk terminates at node *Y.* Note that by definition a BIwalk terminates if and only if it picks the same node twice, i.e., *Xj = Xj+* and in this case we denote *Y* = -Xj+i- For notational convenience we define *XQ* = #o £ *V,* so for any *x* € V, W(xo —• #) = |V|- Note that there is a unique BIwalk between every pair of nodes in *V,* since there is a unique path between every pair of nodes in the tree and the BIwalk only travels over edges in the tree.

Say the BIwalk moves from node *z* to node *x* at the i^{th} step, i.e., *X{-* = *z* and *X{* = *x.* Then the probability that the BIwalk moves to a node y G *V* at the $(i + l)^{st}$ step is given as:

$$P[X_{i+1} \overset{prv}{} = y \backslash X_{\{} \overset{\{Y}{=} {}_{Xr}Xi\text{-}! \overset{Y}{} = z] \overset{,}{=} i \cdot \begin{cases} JO & ify\hat{}\{T_T(x)U\{x\}\}ory = z \\ \frac{w(x\_y)}{I\ W(z\hat{}x)} & omerwise \end{cases} \quad (2)$$

If y = x_y i.e., re chooses itself, then by definition the BIwalk terminates at *x* and *Y* = *x.* It is easy to see from Equation 2 that the BIwalk takes a maximum of tmax steps to terminate where tmax = "Diameter of T". We now prove that the BIwalk samples nodes from *V* (nodes in the tree *T)* uniformly at random.

Theorem 1. *For every* BIwalk, *P[Y = xuut] = l/\V\forall xuut € V.*

Proof: We prove this claim by induction on the size of the tree, *|V\.* For the base case *\V* = 1, the claim holds trivially since x^st is the only node in the tree (by assumption) and so *P[Y* = xiast] = 1.

Assume the claim holds for all trees of size up to fc, i.e., for all trees T = (V', *ET)* such that *|V* ≤ *k.* We prove that it holds for |V| = fe + 1. Say the BIwalk starts at some node x^f E V, i.e., *X* = *x'.* Then there are two possible cases: (i) x^f = xiast- From Equation 2 the probability that the BIwalk terminates at x^f given that it starts at *x'* is *P[Y = x' \ X* = x^f,X_0 = x_0 *g V] = 1/\V*$_9$ since by definition *W*(x^f -> x^1) = 1 and *W(XQ* -> x^1) = |V|. (ii) *x'* ^ *x^-* Since xiast € *V'Qoy* assumption), xiast must be in a component *C*(x^r —> *y)* for some y G *TT{X'').* Then from Equation 2 and the definition *W(XQ* -> x^1) = *\V\,* the probability that the BIwalk enters the component *C*(x^f -> y), i.e., steps from x^f to y is given by:

$$P[X_2 = y \backslash X_1 = x \backslash X_0 = x_0 ? V\} = \frac{{}_V W(x' \text{—>} {}_U y)}{|F|} \quad (3)$$

Note that *C(x'* -> y) is a tree of size at most fc, since |V| = *k* + 1, *x'* G *V* and x^f is not contained in *C(x'* -» y). So by assumption once the BIwalk enters the component *C{x** -> y), it terminates at xiast with probability

$$P[Y = \text{xiast} \mid \text{BIwalk reaches y}] = \frac{1}{w \backslash x \text{ —} ty)}r \quad (4)$$

Node y is in the path from x^1 to xiast and there is a unique BIwalk over the path between every pair of nodes. Therefore, the probability that the BIwalk terminates at xiast when x^1 ^ xiast and $x\backslash\hat{}_x$ is in the component *C{*x^f *-> y)* for some y € IY(x'), is given by:

$$P[Y = \text{xiast}] = P[Y = \text{xiast} \mid \text{BIwalk reaches y}] \text{ x P[BIwalk reaches y]}$$
$$- \frac{1}{W(x' \text{->} y)} \text{ x } \frac{W(x^f \text{->} y)}{|V|} - " \frac{1}{|F|}$$

D

53 Reducing message complexity

The mechanisms described in Section 5.2 assume that each node *x* in the tree *T* knows the weight *W(x -» y)* for each neighbor *y* G *TT(X).* At the start of the execution, this can be achieved by an initial messaging round. However, once all the weights are known, the addition or removal of a node would require multicasting this information to keep the weights updated at all nodes. This is not acceptable due to the large message complexity of multicast. Furthermore, if multicast is being employed then a trivial solution for uniform sampling from the tree exists: the joining node multicasts its arrival and all existing nodes reply with their identities allowing the new node to choose neighbors uniformly at random from the set of all existing nodes.

Our goal is to sample nodes uniformly from the tree using an algorithm that requires a much lower messaging cost than multicast. To achieve this we modify the mechanism described in Section 5.2 as follows: To choose a node uniformly at random from the tree, a node *x* first sends a request called BI request to the root of the tree. The root node then starts a Blwalk on behalf of *x.* As before, if this Blwalk terminates on a node y, then *y* adds *x* to *Tc{y)* and *x* adds *y* to *TG{X).* Theorem 1 proves that irrespective of where this Blwalk originates from, it chooses *y* uniformly at random.

To understand the effects of this minor change, we first note that Equation 2 can also be expressed as:

JO **ify?{r$_T$(x)U{x}}ory = z**

P[Xi+i =y\Xi=x, Xi-$_X$ = z] = < *w(x-△y)* (x→u) otherwise

U L$^{T(x),u≠}$ \ > **omerwise**

Thus to compute the transition probabilities, a node *x* that is currently hosting a Blwalk needs to know the weights of all of its neighbors *u* £ *TT{X)* except the neighbor *z* where the Blwalk came from. In context of the new mechanism this implies that each node only needs to know the weights of its children and not the parent, since the Blwalk always comes from the parent—the Blwalk originates at the root and is irreversible. Therefore, a join or leave operation at node *x,* i.e., a node joins as a child of *x* or some child of *x* leaves the tree, now requires updating the weights only at nodes that are in the path from *x* to the root. This takes only O(logn) messages, a substantial improvement to the multicast required earlier.

5.4 Load balancing

The optimization described in Section 5.3 reduces message complexity considerably for each update but increases the load on the root as every Blwalk originates at the root. We reduce this load by interleaving BI walks with MD walks that run on the expander. Our algorithm constructs the expander incrementally, initially consisting of a small set of nodes and growing in size as new nodes join the expander by sampling enough neighbors from the tree. We say a node *x* is an *expander node* if *\TG{X)* >̲ *d* — e. Once an expander is constructed, nodes can be sampled from the expander using MD walks, in addition to sampling from the tree using Blwalks.

MD walks are a good match to our setting because they have a uniform stationary distribution even on irregular graphs (our expander is an irregular graph), the maximum degree of the expander graph is known to be dmax = *d* + e, and the mixing time is small due to high expansion. For our application, MD walks mix sufficiently in 5 * log(m) steps, where *m* is the number of expander nodes; a detailed analysis of mixing times on different graphs appears in [3]. Nodes can estimate the logarithm of expander size using only local information through mechanisms described in [14]. The main assumption in [14] is that a new node joining the network has a randomly chosen existing node as its first contact point. This fits well with our construction as the expander neighbors are chosen uniformly at random anyway.

Using MDwalks in our system, however, can be problematic in two cases: First, MDwalks sample from a uniform distribution only if the expander is sufficiently large. Second, if the tree contains many nodes that are not expander nodes—e.g., if they just joined the tree or if their neighbors crashed—then the MDwalks will only be sampling from a subset of nodes, since MDwalks only sample from the expander. To address these issues, we develop a "throttling mechanism" shown in Figure 2 that uses more MDwalks as the tree becomes large and more stable—a large, stable tree implies a large expander covering most nodes in the tree. When the tree is large, there are more nodes in path to the root and thus a higher probability of starting an MDwalk (lines 7 and 8). When the tree is stable most nodes are expander nodes and so MDwalks are not interrupted (lines 9–12). An MDwalk stepping on a node that is not an expander node implies that there might be a non-negligible fraction of such nodes in the tree. Hence, in this case the MDwalk is interrupted and a special request BIrequest$'$ is deterministically sent to the root that results in a BIwalk (lines 13–16).

```
Every node $x \in V$ executes the following:
Initialization (addendum to Figure 1):
1.  set parent to $x$'s parent in $T$

Upon receiving (BIrequest : $y$):
2.  if $x$ is root
3.      send (BIwalk : $y$) using Eq. 2
4.  else if $|\Gamma_G(x)| < d - \epsilon$
5.      send (BIrequest : $y$) to parent
6.  else
7.      with prob. $p$, send (BIrequest : $y$) to parent
8.      with prob. $1 - p$, send (MDwalk : $y$) using Eq. 1

Upon receiving (MDwalk : $y$):
9.  if $|\Gamma_G(x)| < d - \epsilon$
10.     send (BIrequest$'$ : $y$) to parent
11. else
12.     send (MDwalk : $y$) using Eq. 1

Upon receiving (BIrequest$'$ : $y$):
13. if $x$ is root
14.     send (BIwalk : $y$) using Eq. 2
15. else
16.     send (BIrequest$'$ : $y$) to parent
```

Figure 2: Interleaving MDwalks with BIwalks to reduce load on the root

We note that our algorithm cannot add or remove undirected edges to the expander graph instantaneously due to the distributed setting. This could be done using some global locking mechanism but at a considerable performance cost, and therefore we avoid that. As a result our expander has some directed edges, e.g., node x has added y to $\Gamma_G(x)$ but y has not yet added x to $\Gamma_G(y)$. The results concerning MDwalks discussed in Section 3.2 relate to undirected graphs. Therefore, when an MDwalk reaches a node y from a node x such that $x \notin \Gamma_G(y)$, y sends the MDwalk back to x and x chooses another neighbor from the set $\Gamma_G(x) \setminus \{y\}$ according to the transition probabilities in Equation 1. This ensures that MDwalks effectively only step from a node to another node if there is an undirected edge between them.

5.5 Tree maintenance

Maintaining the tree T is not essential when the expander is large, since most BIrequests would result in MDwalks. However, in dynamic scenarios we still need to perform some BIwalks. Since BIwalks move across edges of the tree, we need to maintain the tree as a connected component even when some nodes crash.

When a node x crashes, the parent y of x simply removes the crashed child from $\Gamma_T(y)$ and sends the updated weight to its own parent; similar to the case of a node joining. It would seem that a child z of x also only needs to remove x from $\Gamma_T(z)$ and connect itself as a child of some randomly chosen node in $\Gamma_G(z)$ to keep the tree connected. However, the astute reader would notice that if this randomly chosen node is in the subtree rooted at the crashed node *x*, i.e., in the component *C(y —> x)*, then connecting the children of *x* to this node would still leave the tree partitioned. Therefore, *z* must find a node z^1 € *TG{Z)* such that z^1 *e C(x* -> y), i.e., z^1 is in the component that contains the root of the tree. If *z* cannot find any such node then it would have to re-join the tree. Note that nodes in the subtree rooted at *z* do not have to re-join, they will be connected to the tree through *z*.

Figure 3 shows the algorithm run by node *x* in case the failure detector suspects x's parent to have crashed. The failure detector communicates with *x* through Suspect and Unsuspect messages. To find a neighbor in *TG(X)* that is in the component containing the root, *x* uses special tokens (lines 5 and 6) that traverse the path from neighbors of # in *TG(X)* to the root (line 26). If *x* gets one of these tokens back from the root, it knows that the corresponding neighbor in *TG(X)* has a path to the root, *x* then connects as a child of this neighbor (lines 19-22). To avoid cycles that could be formed using this approach, nodes in path to the root add their identifiers to the token and locally store the identifier of the node that created this token in list, list at a node x^r represents nodes that could possibly connect as a child to a node in the subtree of x^r. To avoid cycles, x^f must not connect as a child of a node that is in the subtree of any node in list. Therefore, *x* discards a token from root if this token and list have a common node identifier. This ensures that the "patched" tree does not contain any cycles. The mechanism is efficient since it only takes O(log *n)* time and sends at most *(d + e)* * logn messages.

```
Every node x EV executes the following:
Initialization (addendum to Figure 2):
1.  list <- 0
2.  suspected «- false

Upon receiving (Suspect: parent):
3.  start timer
4.  suspected <— true
5.  for each y € FG(X)
6.      send (Tok : x, y) to y

Upon receiving (Unsuspect: parent):
7.  suspected <— false
8.  stop timer

Upon receiving (TimerExpired : ):
9.  if suspected = true
10.     re-join the tree, set parent to new parent
11.     suspected «— false

Upon receiving (Tok : i,j):
12. if re is root
13.     send (Tok : t,j) toi
14. else if x = i and (Tok : i, j) is sent by root
15.     if (Tok : i, j) and list have a common id
16.         discard (Tok : i, j)
17.     else if suspected = false
18.         discard (Tok : z, j)
19.     else
20.         parent <— j
21.         notify j and send weight to j
22.         stop timer
23. else
24.     add my id to (Tok : i, j)
25.     add i to list
26.     send (Tok : i,j) to parent
```

Figure 3: Maintaining a connected tree in the presence of crash failures

5.6 Summary

Here we summarize the construction of expander graphs from the tree:

- We construct random (d, e)-regular graphs from a tree. Each node uniformly samples nodes from the tree and makes edges with them. This is done continuously so that the network self-stabilizes after crash failures or periods of biased sampling.

- We use a combination of BIwalks and MDwalks to sample nodes from the tree uniformly at random. BIwalks step across edges of the tree and require nodes to know the "weight" of the tree in the direction of each neighbor. This results in a high messaging cost for each update. To reduce this cost, we start all BIwalks from the root. As the expander grows, we can reduce this load on the root by using MDwalks. MDwalks step across edges of the expander. Our algorithm results in more Blwalks when nodes in the tree are not part of the expander or when the expander is small, since in these cases MDwalks sample from a biased distribution.

- Our algorithm patches the tree after a node crashes to retain the ability to do Blwalks. The patching is done such that the tree stays as a single connected component with no cycles.

6 Simulation results

We present simulation results measuring graph expansion and connectivity under different conditions. These results validate our construction and prove that the resulting graphs are tolerant to node failures. We also show that using the mechanisms described in Section 5.4, the load is equally distributed among all nodes in the tree during stable periods, since most Blrequests result in MDwalks. When the tree is more dynamic thus causing more Blwalks, the load on the root is only a constant fraction higher than the load on other nodes as the number of nodes increases. These two results provide evidence that our system scales well.

Verifying if a graph is an expander is co-NP-complete [5] since it requires verifying the expansion of an exponentially large number of subsets of vertices. However, we can estimate a graph's expansion by computing eigenvalues of some graph-related matrices. Let A be the *adjacency matrix* of the graph, i.e., each element *A{j* represents the number of edges between vertices *i* and *j*. Since we only consider simple graphs so *Ay* E {0,1}. Let *D* be the *degree matrix* of the graph, i.e., *D{j* = 0 for *i* ^ *j* and *Dy* = *\Tc(i)* \ for *i* = *j*. Then the *Laplacian matrix L* is defined as *L* = *D* — *A*. The second smallest eigenvalue A of *L* is related to the graph expansion: a graph is a ^f^—expander, where A is the maximum degree of the graph [2], in our case A = *d* + e. We find the Laplacian matrix of the graph, compute its second smallest eigenvalue and find the expansion. The eigenvalue A also gives a lower bound on the vertex connectivity of the graph. However, we use Kleitman's algorithm [16] to find the exact vertex connectivity of our networks; the algorithm computes the vertex connectivity in reasonable time for small n. We note that the theory behind the relation of eigenvalues to expansion and connectivity only deals with undirected graphs. To use these well studied results we ignore all the directed edges when computing the Laplacian matrix of our graph. Therefore, the expansion and vertex connectivity results reported here are pessimistic in the sense that our graphs actually have more edges which are not represented in these results and adding more edges can only improve expansion and connectivity.

We developed a round-based simulator in Java. The simulator sets up an initial topology by constructing a random tree containing *d* + 1 nodes. The simulator constructs (d, e)-regular random graphs with e = d/2 overlayed on this random tree using mechanisms described in earlier sections, riadd nodes are added to the random tree after every Tadd rounds until the total number of nodes in the tree becomes n. Each of the riadd nodes is added as a child to an existing node chosen from a distribution that picks more recently added nodes with a higher probability. This is done so that the experiments measuring the load on each node are not affected due to a node having a lot more children than other nodes. We want to see the load caused by the specifics of our algorithm and not due to such anomalies in the tree. To simulate crash failures, we remove ^move nodes, chosen uniformly at random from the tree, after every T_{rem}ove rounds. Nodes in the tree send Blrequests to their parents every Twaik rounds. Blrequests are forwarded by expander nodes to their parents with probability *p* = 0.5 (the choice is arbitrary, smaller *p* will obviously reduce load on root) and otherwise they initiate MDwalks on the expander. We specify values for n, 7iadd> ^remove, d, Tadd? r_{remove}

and Twaik for different experiments as these are all tunable parameters in our system.

Figure 4 plots the graph expansion and connectivity for different values of $\wedge^{alk}_{add}$. In this experiment we use n = 200, d = 15, n^d = 20, Tadd = 200 and values 10,20,30 and 50 for Twaik. We compute the expansion and connectivity of the graph every single round. The graph plots expansion and connectivity after the first 100 nodes have already been added (only to make the figure more visible). Each point in the plot is a mean of 30 tests, each starting from a new random tree. Expansion and connectivity are determined for all nodes in the tree, not just the expander nodes so that we can see the time it takes for the system to self-stabilize. When new nodes are added to the tree, expansion and connectivity go down to 0 since the new nodes do not have any neighbors in the expander yet. A larger *ijPj*^ ratio implies that nodes look for random neighbors slowly while the graph is changing fast. This results in the graph taking a longer time to stabilize to better expansion and connectivity as shown in the figure.

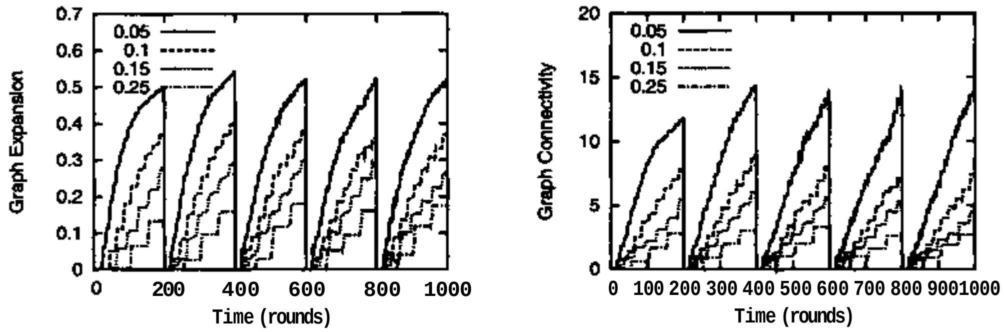


Figure 4: Expansion and Connectivity for various values of $\frac{T_{walk}}{T_{add}}$. Shows how quickly new nodes join the expander after being added to the tree.

Figure 5 shows network properties in the presence of crash failures. We simulate perfect failure detection at each node for its neighbors in the tree. Nodes run the algorithm from Section 5.5 to connect the tree after their neighbors crash. For this experiment we use n = 150, d = 15,n$_{rem}$ove = 10, Twaik = 10 and T$_{re}$move = 300. All 150 nodes are first added to the tree and the expander is allowed to stabilize for 1000 rounds (this period is not shown in the plot). We then start measuring the expansion and connectivity every single round and remove 10 nodes after every 300 rounds until we are left with 100 nodes. Each point in the plot is a mean of 15 tests, each test starting from a different random tree. The figure shows that crashed nodes have almost no impact on graph expansion and affect vertex connectivity very slightly. This plot like Figure 4 also shows the self-stabilizing nature of our network during stable periods.

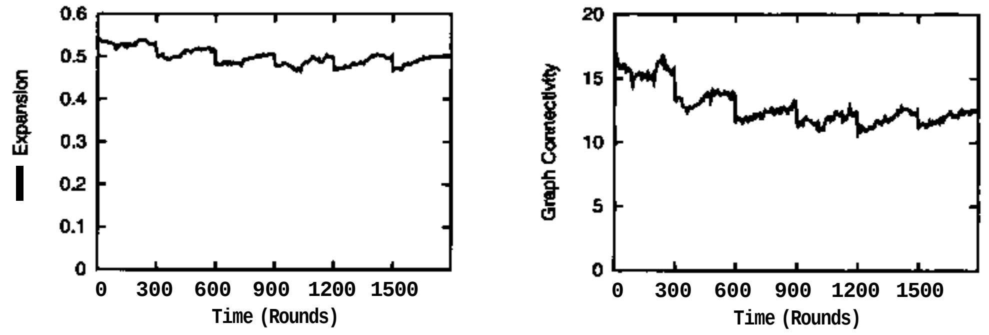


Figure 5: Expansion and Connectivity as nodes in the network crash

Figure 6 plots graph expansion and connectivity against different values of *d.* We use *n* = 200, n_{add} = 20, Tadd = 100, TWaik = 10 and varied *d* = 4,7,10,13,16,19,22 and 25. For each value of d, we waited 1500 rounds after adding all nodes to the graph (to give it enough time to stabilize) and then measured expansion and connectivity at 30 different points separated by 1000 rounds each. The plot shows mean expansion and connectivity of these 30 rounds for each *d.* As shown in the figure, expansion and connectivity increase with *d.* Our graphs achieve reasonable fault-tolerance even for small values of *d* like *d* = 10.

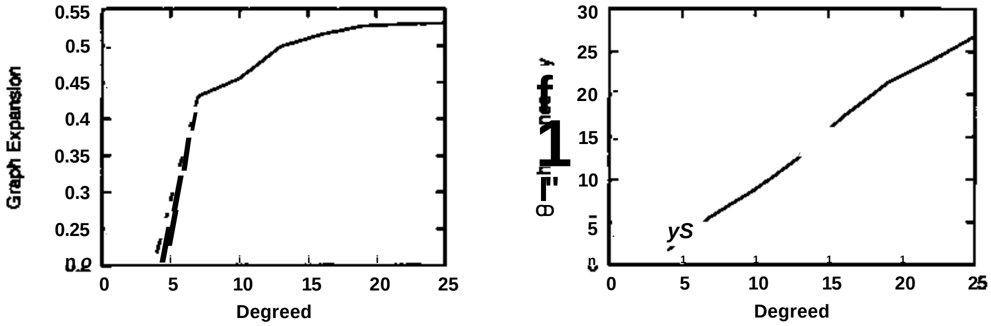
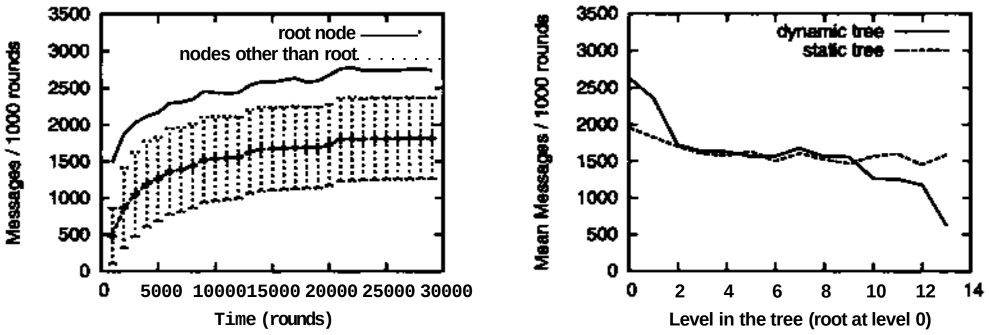


Figure 6: Expansion and Connectivity for various values of d. The actual degrees are between *[d* + e, *d* — e].

Figure 7-(a) compares the load (number of messages handled) on the root node with the mean load on other nodes (and the standard deviation) in the tree. For this experiment we used n = 3000, *d* = 15, nadd = 10, Tadd = 100 and Twaik = 10. We start measuring the load from the first round by counting the number of messages received by each node every 1000 rounds. We stop the experiment when all nodes are added to the system, i.e., after 30,000 rounds. Each point in the plot is a mean of 30 tests with each test starting from a different random tree. The dashed curve plots the mean load seen by all nodes except the root along-with the standard deviation. This standard deviation is high since nodes closer to the root have a higher load than nodes closer to the leaves. The plot shows a slight increase in the load on all nodes as the number of nodes in the graph increases. This is because the M Dwalks run longer as the number of nodes increases (see Section 5.4). However, this effect becomes less visible when the number of nodes is large. Also note that the load on the root remains only a constant fraction more than the load on other nodes as the number of nodes increases. This constant can also be controlled using the parameter p. We use *p* = 0.5 in all experiments, a smaller value would reduce the constant difference between the load on root and other nodes. For lack of space we do not present these results here.

Figure 7-(b) plots the load on the nodes against the level of these nodes in the tree, with root at level 0. We use the same values for all the parameters as in Figure 7-(a). For the dynamic case, i.e., when nodes are being added to the graph, load on the nodes is measured every 1000 rounds like before and the mean load for each node in computed over all rounds in which the node was in the tree. This mean load for each node is used to compute the mean load for each level in the tree which is plotted. For the static case, we start measuring load on the nodes after all nodes have been added to the tree. The expander is given some time to stabilize (500 rounds) from the last addition and then 15 different measurements are taken, one every 1000 rounds, of the load seen by all nodes. Mean of these 15 measurements is computed for each node and the result is used to compute the mean load for each level. For the dynamic case, i.e., when nodes are being added to the tree every Tadd period, the load increases as we go up the tree. This is a result of more BIwalks due to the dynamic scenario. For the static case the load is distributed almost uniformly across different levels in the tree, since most walks are MDwalks. These graphs show the graceful degradation in our algorithms, i.e., load on the root and higher levels in the tree increases as the tree becomes more dynamic and remains uniform in static scenarios.



(a)

(b)

Figure 7: Load on root node compared to other nodes

7 Conclusions

We present a distributed construction of an expander graph given a tree structure. Following a pragmatic approach, we construct an almost-regular random graph that is much easier to construct and maintain than a strictly-regular random graph but achieves the required connectivity and expansion properties. Our construction tolerates crash failures under realistic assumptions using a self-healing approach. We also present a novel distributed technique to sample nodes perfectly uniformly at random from a tree even when the number of nodes in the tree is small. This technique is used to bootstrap the process of generating random almost-regular graphs. Our simulation results show that the graphs generated have high expansion and connectivity and our construction yields almost constant load on all nodes under various dynamic and static conditions.

References

[1] M. Ajtai, J. Koml'os, and E. Szemer'edi. An $O(n \log n)$ sorting network. *Combinatorica,* 3(1):1—19,1983.

[2] N. Alon. Eigenvalues and expanders. *Combinatorica,* 6(2):83-96,1986.

[3] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Distributed uniform sampling in real-world networks. In *Purdue University, CSD Technical Report (CSD-TR-04-029),* Oct. 2004.

[4] A. Bagchi, A. Bhargava, A. Chaudhary, D. Eppstein, and C. Scheideler. The effects of faults on network expansion. In *Proc. 16th ACM Symposium on Parallel Algorithms and Architectures,* June 2004.

[5] M. Blum, R. M. Karp, O. Vornberger, C. H. Papadimitriou, and M. Yannakakis. The complexity of testing whether a graph is a superconcentrator. *Information Processing Letters,* 13(4/5): 164-167,1981.

[6] B. Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics,* 1(4):311-316,1980.

[7] S. Boyd, P. Diaconis, and L. Xiao. Fastest mixing markov chain on a graph. *SIAM Review,* 46(4):667-689,2004.

[8] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *Microsoft Research, Technical Report MSR-TR-2002-82,*2002.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM,* 43(2):225-267,1996.

[10] J. Friedman. On the second eigenvalue and random walks in random d-regular graphs. *Combinatorica,* 11(4):331-362,1991.

[11] O. Gabber and Z. Galil. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Sciences,* 22(3):407^20,1981.

[12] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proc. Infocom,* Mar. 2004.

[13] A. Goerdt. Random regular graphs with edge faults: Expansion through cores. In *Proc. 9th International Symposium on Algorithms and Computation,* 1998.

[14] K. Horowitz and D. Malkhi. Estimating network size from local information. *Information Processing Letters,* 88(5):237-243,2003.

[15] J. H. Kim and V. H. Vu. Generating random regular graphs. In *Proc. 35th ACM Symposium on Theory of Computing,* June 2001.

[16] D. Kleitman. Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuits and Systems,* 16(2):232-233,1969.

[17] C. Law and K.-Y. Siu. Distributed construction of random expander networks. In *Proc. Infocom,* Apr. 2003.

[18] Y. Liu, X. Liu, L. Xiao, L. M. Ni, and X. Zhang. Location-aware topology matching in p2p systems. In *Infocom,* Mar. 2004.

[19] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *ACM SIGCOMM Conference,* Aug. 2003.

[20] L. Lovasz. Random walks on graphs: A survey. *Combinatorics, Paul Erdos is Eighty,* 2:1—46,1993.

[21] E. K. Lua, J. Crowcrofl, and M. Pias. Highways: Proximity clustering for scalable peer-to-peer networks. In *Proc. 4th Conference on P2P Computing,* Aug. 2004.

[22] P. MacKenzie and M. K. Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing,* 16(4):307-327,2003.

[23] P. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security,* 2(1):1-20,2003.

[24] G. A. Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii,* pages 71-80,1973.

[25] G. Pandurangan, P. Raghavan, and E. Upfal. Building low-diameter p2p networks. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science,* Oct. 2003.

[26] N. Pippenger and G. Lin. Fault-tolerant circuit-switching networks. In *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures,* June 1992.

[27] M. K. Reiter, A. Samar, and C. Wang. Design and implementation of a jca compliant capture protection infrastructure. In *Proc. of the 22nd IEEE Symp. on Reliable Dist. Syst.,* Oct. 2003.

[28] M. Sipser and D. A. Spielman. Expander codes. *IEEE Transactions on Information Theory,* 42(6):1710-1722, 1996.

[29] A. Steger and N. Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing,* 8(4):377-396,1999.