Distributed Construction of a Fault-Tolerant Network from a Tree

Michael K. Reiter*† Asad Samar* Chenxi Wang*

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. The tree overlayed with this logical expander is a useful structure for distributed applications that require the intrinsic "treeness" from the topology but cannot afford any obstruction in communication due to failures. 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 node failures, the expander maintains its own fault tolerance and permits the reformation of the tree. 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

Trees are an important class of data structures that are suitable to many distributed applications. Their acyclic structure allows the use of simple protocols for data sharing and coordination, e.g., key management [19, 29], hierarchical peer-to-peer systems [15] and distributed mutual exclusion protocols [37, 9, 20, 44, 34]. Moreover the hierarchical nature of trees maps directly to many real world applications, e.g., the Domain Name System [33], distributed certification authorities [25] and distributed directory protocols [11]. Even with these features, trees are not pervasive in many networked applications due mainly to their poor fault tolerance. Node and link failures are considered common-case in distributed systems and a single such failure can partition the tree, crippling the protocol that uses this structure.

This work was supported in part by National Science Foundation award CCR-0208853.

*Electrical & Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA, USA

†Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA

Apart from applications that are inherently tied to the hierarchical and acyclic properties, trees also arise naturally in applications where a node "joins" the system by contacting some node already present in the system: the joining node then becomes a child of the node it contacts for entry, resulting in a tree. Examples include many service discovery protocols where a new server joins the system by contacting an existing server [43, 10, 24], multicast acknowledgement tree construction protocols [39] and several other systems that use the expanding ring technique [6] to locate and join an existing system. Ideally, these applications should be able to use the simple joining procedure and still achieve unimpeded communication among the members when faults occur.

In this paper we address the challenges posed by these two classes of distributed applications by presenting an algorithm that efficiently builds a logical fault-tolerant communication network overlayed on a distributed tree structure. Applications that do not require the "treeness" property can allow new nodes to join simply by attaching to one existing node and run all communication on the fault-tolerant overlay constructed from the tree. For applications that require the acyclic and hierarchical nature of trees, we present distributed algorithms that use the overlay to keep the underlying tree connected in the presence of faults.

The overlay network constructed by our distributed algorithm is an expander. Expanders are an important class of graphs that have found applications in the construction of error correcting codes [41], de-randomization [1], and in the design of fault-tolerant switching networks [36]. The fault tolerance of expanders [17, 4] is precisely what motivated their use in this research. Our algorithm starts with nodes connected in a tree and proceeds to add edges to achieve an expander. Since explicit constructions of expanders are generally very complex, we 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 [13].

The contributions of this work rest primarily in three features. First, our algorithm is completely distributed. Though expander graphs have been studied extensively, distributed construction of expander networks remains a chal-

Proceedings of the 2005 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)
0-7695-2463-X/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

lenging problem. Our algorithms use only local information at each node that consists of the identities of the node's neighbors in the tree. 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 with low message complexity.

Second, our algorithm adapts to node joins, leaves and failures. Previous attempts at distributed construction of random expanders [26, 16] 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; e.g., most of these constructions require nodes to propagate their state to other nodes before leaving the network. We follow a more pragmatic approach, in that we only require that nodes have "close" to d neighbors. In doing so we define a new class of random graphs which we call (d, ϵ)-regular random graphs. These graphs give us more flexibility in dealing with the dynamic nature of our network, while still achieving fault tolerance. One consequence of using this approach is that we do not require nodes to notify others when leaving the network, thereby accommodating failures. To the best of our knowledge, distributed construction of expanders that adapt and regain fault tolerance even when a large fraction of nodes fail, has not been addressed previously.

Finally, we present a novel distributed algorithm that uses the overlay expander to keep the underlying tree connected in the presence of faults. This algorithm works on a "best-effort" basis—in most cases the algorithm is able to successfully patch the tree when nodes fail, however, in the unlikely event of a large fraction of nodes failing simultaneously or in some corner cases like the failure of the root node, the algorithm might not succeed. In these cases we require some of the nodes to re-join the tree using the default application-specific mechanism.

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* [31], 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 [30], giving rise to a tree structure in which the new server is a child of the server that delegated to it [38]. 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 in the tree of authorized servers despite server failures (possibly attacker-induced), we approached the problem that we address in the present paper.

The remainder of this 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 along with related proofs. Tree maintenance using the expander is discussed in Section 6. Section 7 presents simulation results. We conclude in Section 8.

2 Related work

Fault-tolerant tree structures were first introduced in the context of multiprocessor computer architectures as X-Trees [12] and Hypertrees [18]. Fault tolerance was not the primary goal of this research. As a result, these structures impose other constraints that may not be reasonable in our target applications, e.g., X-tree [12] assumes a complete binary tree and tolerates only a single node failure. Furthermore, distributed constructions of X-tree and Hypertree are not known.

Expander graphs are a well studied design for fault-tolerant networks. Both randomized [42, 22] and explicit [32, 14] constructions of expanders have been known for some time. However, little has been done to construct expander networks in a distributed setting.

Law and Siu [26] presented a distributed construction of expander graphs based on $2d$-regular graphs composed of d Hamiltonian cycles. However, to sustain expansion of the graph in the event of nodes leaving the system, they require that a leaving node send its state to some other node in the expander. Therefore, this approach cannot tolerate node failures. Furthermore, their algorithm 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 from a sufficiently large number of nodes.

Gkantsidis, Mihail and Saberi [16] extend the mechanisms presented by Law and Siu [26] to construct expanders more efficiently. 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 a daemon. Thus, as noted in [16], this system is only "weakly decentralized". In addition, node

IEEE COMPUTER SOCIETY

departures are handled as in [26], requiring special messages to be sent by nodes leaving the system.

Pandurangan, Raghavan and Upfal [35] 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 employs a centralized server, known to all nodes in the system, that helps nodes pick random neighbors.

Loguinov et al. [27] present a distributed construction of fault resilient networks based on de Bruijn graphs that achieve good expansion. However, they also require nodes leaving the system to contact and transfer state to existing nodes and thus cannot tolerate 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 $S(n,d)$ 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* (we formally define expansion in Section 4) with high probability [13].

Configuration model [7] is the standard method for generating random d-regular graphs on n nodes $v_1, v_2, ..., v_n$, though not in a distributed setting. In this model each vertex is represented as a set containing d elements, called *points*, resulting in n such sets $\gamma(v_1), \gamma(v_2), ..., \gamma(v_n)$. A *perfect matching* of these nd points is a set of $\frac{nd}{2}$ pairs of points such that every point appears in exactly one pair. 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 v_i and v_j if and only if the perfect matching pairs a point in $\gamma(v_i)$ to a point in $\gamma(v_j)$. This model allows self loops (pairing points from the same set) and parallel edges (more than one pair from the same two sets) and is very inefficient if the goal is to construct a *simple graph*, i.e., one without self loops and parallel edges. A refinement [42] 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 $d \leq n^{1/3-\epsilon}$, for any positive constant ϵ [22]. In Section 5, we extend this model to the distributed setting for building expander graphs.

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 $n \times n$ matrix where each element M_{ij} specifies the probability with which the random walk moves from node i to node j in one step. Let π_t be a vector such that $\pi_t[i]$ is the probability with which the random walk visits vertex i at step t. Then $\pi_{t+1} = \pi_t M = \pi_0 M^{t+1}$. A vector π is called the *stationary distribution* of the random walk if $\pi = \pi M$, 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 [28]. *Mixing time* is the time required for the random walk to reach its stationary distribution and it depends on the expansion of the graph: the walk reaches the 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 $\Gamma_G(x)$ denote the set of neighbors of node x in graph G. 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 $\Gamma_G(x)$ with probability $1/|\Gamma_G(x)|$. The stationary distribution of a simple random walk on a regular graph is uniform, i.e, $\pi = \frac{1}{n}[1, 1, ..., 1]$. 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, 8]) to sample uniformly at random from an irregular graph G with maximum degree $d_{\max}$ is to run a random walk on G that takes a step from node x to node y with probability:

$$P_{xy} = \begin{cases} \frac{1}{d_{\max}} & \text{if } y \neq x \text{ and } y \in \Gamma_G(x) \\ 1 - \frac{|\Gamma_G(x)|}{d_{\max}} & \text{if } y = x \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

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: First in a dynamic distributed system it is often difficult to estimate the maximum degree of the graph. Second, 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 undirected tree and denoted as $T = (V, E_T)$. The vertex set of the tree and the overlay expander is the same but their edge sets differ, hence the subscript. For any subset $S \subset V$ we define the set of *neighbors* of S in T as $\Gamma_T(S) = \{y \in V \mid \exists x \in S, (x, y) \in E_T\}$. 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 experience fail stop [40] failures, thus failure of a node can be detected by other nodes in the system. Our algorithms are designed independent of a particular fault distribution, however, our experiments use a random distribution of faults. See [4] for a detailed analysis of how expanders behave under different fault distributions.

We present some notation used to define expander graphs.

Definition 1. *Given a graph $G = (V, E_G)$, the vertex boundary $\partial_G(S)$ of a set $S \subset V$ is $\partial_G(S) = \{y \in V \setminus S \mid \exists x \in S,\ (x, y) \in E_G\}$.*

Definition 2. *A graph $G = (V, E_G)$ is an α-expander if for every subset $S \subset V$ of size $|S| \leq |V|/2$, $|\partial_G(S)| \geq \alpha|S|$, for some constant $\alpha > 0$.*

Our goals can be summarized as follows: Construct an expander graph with the vertex set V using a distributed algorithm that scales well. New nodes should be able to join the expander with a low messaging cost even when the expander is very large. In the event of node failures, the expander should "self heal" to regain its fault tolerance and the partitioned underlying tree should be patched to a single connected component.

5 Distributed expander construction

Our approach is to construct a random graph among vertices in V (nodes in the tree) 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 distributed system with dynamic membership than a d-regular random graph, while still achieving comparable expansion.

5.1 Random almost-regular graphs

We say a graph is (d, ϵ)*-regular* if the degrees of all nodes in the graph are in the range $[d - \epsilon, d]$. Then a (d, ϵ)*-regular random graph*, denoted $G_{n,d,\epsilon}$, is a (d, ϵ)-regular graph that contains a subgraph chosen uniformly at random from the set $S(n, d-\epsilon)$—the set of all $(d-\epsilon)$-regular graphs on n nodes. Therefore, the expansion of $G_{n,d,\epsilon}$ is bounded from below by the expansion of $G_{n,d-\epsilon}$, since adding edges to a graph does not decrease its expansion.

Our distributed construction builds (d, ϵ)-regular random graphs by approximating the refinement [42] of the configuration model (see Section 3.1) as shown in Figure 1. $\Gamma_G(x)$ is the set containing x's neighbors in the overlay expander. Nodes are sampled from the tree (line 4)—using mechanisms discussed later—and added to this set, maintaining a maximum of d neighbors (line 3 and lines 9–12). We avoid self-loops and parallel edges (lines 6 and 7). Upon detecting the failure of an expander neighbor, x removes this node from $\Gamma_G(x)$ (line 16).

```
Every node x ∈ V executes the following:
Initialization:
1.  Γ_G(x) ← ∅

Main:
2.  repeat forever
3.      if |Γ_G(x)| < d
4.          uniformly sample node y from V
5.          send (Add : y) to x

Upon receiving (Add : y):
6.  if y = x or y ∈ Γ_G(x)
7.      do nothing
8.  else
9.      if |Γ_G(x)| = d
10.         pick z from Γ_G(x) at random
11.         remove z from Γ_G(x)
12.         send (Remove : x) to z
13.     add y to Γ_G(x)
14.     send (Add : x) to y

Upon receiving (Remove : y):
15. remove y from Γ_G(x)

Upon receiving (Failed : y)
16. remove y from Γ_G(x)
```

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

Using (d, ϵ)-regular random graphs allows us to avoid complicated mechanisms that synchronize the state of departing nodes with nodes in the network in an attempt to maintain exactly d neighbors. Instead, we allow nodes to leave without announcing their departure and ignore periods where some nodes may have less than d neighbors. A large number of simultaneous failures can result in some nodes having degrees even less than $d - \epsilon$, but the fault tolerance of the expander will ensure that most nodes remain connected in a component that has high expansion. This allows nodes with low degrees to recover "quickly". We present results related to the convergence rate of the expander under

0-7695-2463-X/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

different conditions in Section 7.

These mechanisms reduce the problem of constructing $G_{n,d,\epsilon}$ to that of a node $x \in V$ choosing another node uniformly at random from the tree (line 4), i.e., with probability $1/|V|$. Such a sampling procedure could be used by nodes to construct and maintain $G_{n,d,\epsilon}$ as described above.

5.2 Biased irreversible random walks

For a node, choosing another node uniformly at random from the tree is challenging because the structure is a tree (and not a random graph for example) and because 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): x knows the size of the subtree rooted at each of its children and x knows the number of nodes in the tree that are not in the subtree rooted at x—this is the number of nodes in the direction of x's parent. Then, to choose a node uniformly at random from the tree, x starts a *biased irreversible random walk*, BIwalk. At each step, the BIwalk 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 BIwalk *terminates*. The node where the BIwalk 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 BIwalk samples nodes uniformly at random from the tree when the tree is static.

Let (x, y) be an edge in E_T (E_T is the edge set of T) and $F(V, E_T \setminus \{(x, y)\})$ be the forest containing two components formed by removing (x, y) from T. Then, we define $C(x \lhd y)$ to be the component of F that contains node y. The '$\lhd$' notation captures the intuition that this is x's view of the tree in the direction of its neighbor y. Let V' denote the vertex set of $C(x \lhd y)$, then $W(x \lhd y) = |V'|$. Intuitively, $W(x \lhd y)$ represents x'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 $W(x \lhd y) = |V|$ if $x \notin V$ and $y \in V$ (the view from outside the tree), and $W(x \lhd y) = 1$ if $x = y$ (the view when x looks down at itself).

We denote a BIwalk as a sequence of random variables $X_1, X_2, ..., Y$, where each X_i represents the node that initiates the ith step of the BIwalk (X_1 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., $X_j = X_{j+1}$ and in this case we denote $Y = X_{j+1}$. For notational convenience we define $X_0 = x_0 \notin V$, so for any $x \in V, W(x_0 \lhd x) = |V|$. 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 $(i-1)^{st}$ step, i.e., $X_{i-1} = z$ and $X_i = x$. Then the probability that the BIwalk moves to a node $y \in V$ at the ith step is given as:

$$\Pr[X_{i+1} = y \mid X_i = x, X_{i-1} = z] = \begin{cases} \frac{W(x \lhd y)}{W(z \lhd x)} & \text{if } y \in (\Gamma_T(x) \cup \{x\}) \setminus \{z\} \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

If $y = x$, i.e., x 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 $t_{\max}$ steps to terminate, where $t_{\max}$ is the 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, $\Pr[Y = x_{last}] = 1/|V|$ *for all* $x_{last} \in V$.

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_{last} is the only node in the tree (by assumption) and so $\Pr[Y = x_{\text{last}}] = 1$.

Assume the claim holds for all trees of size up to k, i.e., for all trees $T = (V, E_T)$ such that $|V| \leq k$. We prove that it holds for $|V| = k+1$. Say the BIwalk starts at some node $x_1 \in V$, i.e., $X_1 = x_1$. Then there are two possible cases:

1. $x_1 = x_{\text{last}}$. From Equation 2 the probability that the BIwalk terminates at x_1 given that it starts at x_1 is $\Pr[Y = x_1 \mid X_1 = x_1, X_0 = x_0 \notin V] = 1/|V|$, since by definition $W(x_1 \lhd x_1) = 1$ and $W(x_0 \lhd x_1) = |V|$.

2. $x_1 \neq x_{\text{last}}$. Let y be the neighbor of x_1 such that x_{last} is in the component $C(x_1 \lhd y)$. Then from Equation 2 and the definition $W(x_0 \lhd x_1) = |V|$, the probability that the BIwalk enters the component $C(x_1 \lhd y)$, i.e., steps from x_1 to y is given by:

$$\Pr[X_2 = y \mid X_1 = x_1, X_0 = x_0 \notin V] = \frac{W(x_1 \lhd y)}{|V|} \quad (3)$$

Note that $C(x_1 \lhd y)$ is a tree of size at most k, since $|V| = k+1$, $x_1 \in V$ and x_1 is not contained in $C(x_1 \lhd y)$. So by assumption once the BIwalk enters the component $C(x_1 \lhd y)$, it terminates at x_{last} with probability

$$\Pr[Y = x_{\text{last}} \mid \text{BIwalk reaches } y] = \frac{1}{W(x_1 \lhd y)} \quad (4)$$

0-7695-2463-X/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Node y is in the path from x_1 to x_{last} and there is a unique BIwalk between every pair of nodes. Therefore, the probability that the BIwalk terminates at x_{last} when $x_1 \neq x_{\text{last}}$ and x_{last} is in the component $C(x_1 \triangleleft y)$ for some $y \in \Gamma_T(x_1)$, is given by:

$$\begin{aligned}\Pr[Y = x_{\text{last}}] &= \Pr[Y = x_{\text{last}} \mid \text{BIwalk reaches } y] \times \\ &\quad \Pr[\text{BIwalk reaches } y] \\ &= \frac{1}{W(x_1 \triangleleft y)} \times \frac{W(x_1 \triangleleft y)}{|V|} = \frac{1}{|V|} \quad \square\end{aligned}$$

5.3 Reducing message complexity

The mechanism described in Section 5.2 assumes that each node x in the tree T knows the weight $W(x \triangleleft y)$ for each neighbor $y \in \Gamma_T(x)$. 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 to 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.

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 BIrequest to the root of the tree. The root node then starts a BIwalk on behalf of x. As before, if this BIwalk terminates on a node y, then y adds x to $\Gamma_G(y)$ and x adds y to $\Gamma_G(x)$. Theorem 1 proves that irrespective of where this BIwalk originates (from x or from root), 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:

$$\Pr[X_{i+1} = y \mid X_i = x, X_{i-1} = z] = \begin{cases} \frac{W(x \triangleleft y)}{1+\sum_{u \in \Gamma_T(x), u \neq z} W(x \triangleleft u)} & \text{if } y \in (\Gamma_T(x) \cup \{x\}) \setminus \{z\} \\ 0 & \text{otherwise} \end{cases}$$

Thus to compute the transition probabilities, a node x that is currently hosting a BIwalk needs to know the weights of all of its neighbors $u \in \Gamma_T(x)$ except the neighbor z where the BIwalk came from. In the 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 BIwalk always comes from the parent—the BIwalk 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 $O(\log n)$ messages assuming a balanced tree, 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 BIwalk originates at the root. We reduce this load by interleaving BIwalks with MDwalks (see Section 3.2) 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 $|\Gamma_G(x)| \geq d - \epsilon$. Once an expander is constructed, MDwalks can be used to sample from the set of expander nodes.

MDwalks 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 and the mixing time is small due to high expansion. For our application, MDwalks 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 [21]. The main assumption in [21] 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.

Using MDwalks in our system, however, raises two issues: 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 several of their neighbors failed resulting in less than $d - \epsilon$ neighbors—then the MDwalks will only be sampling from a subset of nodes, since MDwalks only sample from the expander nodes. To address these issues, we develop a "throttling mechanism" shown in Figure 2 that results in more MDwalks as the tree becomes large and stable—a large, stable tree implies a large expander covering most nodes in the tree. Nodes send BIrequests along the path towards the root so the root can start a BIwalk on their behalf, as described in Section 5.3. However, upon receiving a BIrequest from its child, an expander node forwards this request towards the root only with probability p (lines 7 and 8). With probability $1 - p$, the expander node starts an MDwalk (lines 9 and 10) on behalf of the node that initiated the BIrequest. An MDwalk

0-7695-2463-X/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

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 (lines 11 and 12) and a special request BIrequest′ is deterministically sent to the root that results in a BIwalk (lines 19–22). When the tree is large, there are more nodes in the path to the root and thus a higher probability of starting an MDwalk (lines 7–10). When the tree is stable most nodes are expander nodes and so MDwalks are not interrupted (lines 11 and 12).

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

Upon receiving (MDwalk : u):
11. if $|\Gamma_G(x)| < d - \epsilon$
12. send (BIrequest′ : u) to parent
13. else
14. choose y using Eq. 1
15. if $y = x$
16. send (Add : u) to x
17. else
18. send (MDwalk : u) to y

Upon receiving (BIrequest′ : u):
19. if x is root
20. send (BIwalk : u) to y chosen using Eq. 2
21. else
22. send (BIrequest′ : u) to parent

Figure 2: Using MDwalks with BIwalks to reduce root load

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 is therefore avoided. As a result the 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 uniform sampling by MDwalks discussed in Section 3.2 relate to undirected graphs only. 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 Summary

Our construction of an expander from a tree can be summarized as follows:

- We construct (d, ϵ)-regular random graphs from a tree. Each node uniformly samples nodes from the tree and adds them to its neighbor set, maintaining a maximum of d neighbors.
- We use BIwalks to sample nodes uniformly at random from the tree. All BIwalks are started from the root as this requires low message complexity for each update.
- As the expander grows, we can reduce load on the root by using MDwalks. MDwalks step across edges of the expander. Our algorithm results in more MDwalks as the tree grows in size and becomes relatively stable.

Once constructed, the expander can be used by applications for fault-tolerant communication even when their simple joining procedure results in a tree. The other class of distributed applications that employ tree structures to exploit the intrinsic hierarchical and acyclic properties, require the tree itself to be fault-tolerant. We address this in the next section.

6 Tree reconstruction after failures

Keeping the tree connected is essential for applications that need to communicate across tree edges. It is also desirable in the construction of the expander, especially in dynamic scenarios when we need to use BIwalks that run across tree edges. Here we present a distributed algorithm that uses the fault-tolerant expander to "patch" the tree in the event of node failures.

When a node z fails, the parent z' of z simply removes its failed child from $\Gamma_T(z')$ and sends the updated weight to its own parent (except when z' is root), similar to the case of a node joining. It would seem that a child x of z also only needs to remove z from $\Gamma_T(x)$ and connect itself as a child of some randomly chosen expander neighbor in $\Gamma_G(x)$. However, if this randomly chosen neighbor is in the subtree rooted at the failed node z, i.e., in the component $C(z' \triangleleft z)$ (shown by the small triangle in Figure 3), then connecting x (and any other children of z) to this node would still leave the tree partitioned. Therefore, x must find an expander neighbor $y \in \Gamma_G(x)$ (x's expander neighbors are

IEEE COMPUTER SOCIETY

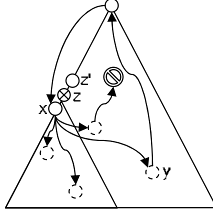

Figure 3: Tree maintenance using the expander. Triangle denotes the tree. Small triangle denotes the subtree rooted at the failed node z. Curved arrows show tokens sent by z's child x to x's expander neighbors denoted by dashed circles.

shown as dashed circles in Figure 3) such that $y \in C(z \triangleleft z')$, i.e., y is in the component that contains the root of the tree.

Our approach to find such a node is to send "tokens" from x to its expander neighbors. Upon receiving such a token, a node forwards the token to its parent in an attempt to reach the root. If such a token does in fact reach the root, it implies that the corresponding neighbor in $\Gamma_G(x)$ (y in Figure 3) is in the component containing the root. The root sends the token back to x and x attaches itself as a child of y.

Figure 4 shows the distributed algorithm run by node x in case it detects that its parent has failed. The mechanism described above provides only a probabilistic guarantee to find a node in the component containing the root. Therefore, x starts a timer (line 2) before sending the tokens. If a suitable candidate for the new parent is not found within the specified timeout period, e.g., because a large fraction of nodes failed or the root itself failed, then x re-joins the tree using the application-specific mechanism (line 5). We presume that the application-specific joining mechanism is more costly, e.g., because it involves manual intervention.

The token $(\mathsf{Tok} : x, y)$ sent by x to its expander neighbor y (line 4) is forwarded along the path from y to the root (line 13), which finally returns the token back to x (lines 6 and 7). x sets y as its parent, unless the parent has already been set to another node, e.g., because the timer expired or because a different token was received from the root earlier (lines 8–11). To avoid complex scenarios that could result in the formation of cycles, a node x must discard tokens of the form $(\mathsf{Tok} : x, y)$, if the token is forwarded to x by a child. In addition, "nonces" should be used to distinguish between tokens sent across different runs of the protocol. We omit these details from the pseudo-code for brevity.

7 Simulation results

We present simulation results measuring graph expansion and connectivity under different conditions. These results validate our expander construction and prove that the resulting graph is tolerant to node failures. We also show

Every node $x \in V$ executes the following:
Upon receiving (Failed : parent):
1. parent $\leftarrow \perp$
2. start timer
3. for each $y \in \Gamma_G(x)$
4. send $(\mathsf{Tok} : x, y)$ to y

Upon receiving (TimerExpired :):
5. re-join the tree, set parent to new parent

Upon receiving $(\mathsf{Tok} : x', y')$:
6. if x is root
7. send $(\mathsf{Tok} : x', y')$ to x'
8. else if $x = x'$ and $(\mathsf{Tok} : x', y')$ is sent by root
9. if parent $= \perp$
10. parent $\leftarrow y'$, update weight at y'
11. stop timer
12. else
13. send $(\mathsf{Tok} : x', y')$ to parent

Figure 4: Tree maintenance in the presence of node failures

that using the mechanisms described in Section 5.4, the load is better distributed among nodes in the tree during stable periods, since most BIrequests result in MDwalks. When the tree is more dynamic thus causing more BIwalks, the load on the root is higher than the load on other nodes roughly by a constant amount, even 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 the second smallest eigenvalue λ of the graph's *Laplacian matrix*: a graph is a $\frac{2\lambda}{2\lambda+\Delta}$-expander, where Δ is the maximum degree of the graph [2], in our case $\Delta = d$. We use Kleitman's algorithm [23] to find the vertex connectivity of the expander. We note that the theory behind these well known results deals only with undirected graphs. Thus, we ignore any directed edges in the expander network when computing its expansion and connectivity. Therefore, the results reported in this section are pessimistic in the sense that our graphs actually have more edges which are not represented here.

We developed a round-based simulator in Java. The simulator sets up an initial topology by constructing a random tree containing $d + 1$ nodes. Nodes construct (d, ϵ)-regular random graph with $\epsilon = d/2$ overlayed on this random tree using mechanisms described in earlier sections. In what follows, n denotes the upper bound on the number of nodes used in the experiment; i.e., if node joins are being simulated then nodes are added until the total number of nodes is n, whereas if node failures are being simulated then nodes

IEEE COMPUTER SOCIETY

are removed starting with an initial set of n nodes. To simulate nodes joining the tree, n_{add} nodes are added to the random tree after every T_{add} rounds until the total number of nodes in the tree becomes n. Each of the n_{add} 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 different nodes are not affected due to a node having many more children than other nodes. To simulate node failures, we remove n_{remove} nodes, chosen uniformly at random from the tree, after every T_{remove} rounds. Nodes in the tree send BIrequests to their parents every T_{walk} rounds if they have less than d neighbors. T_{walk} simulates latency and other factors in real networks. We vary this parameter in some experiments to see the effect of these delays on the convergence rate of our algorithms. BIrequests are forwarded by expander nodes to their parents with probability $p = 0.5$ (the choice is arbitrary, smaller p will obviously reduce load on the root). Expander nodes initiate MDwalks with probability $1 - p$. We specify values for $n, n_{\text{add}}, n_{\text{remove}}, d, T_{\text{add}}, T_{\text{remove}}$ and T_{walk} for different experiments as these are all tunable parameters.

Figure 5 plots the graph expansion and connectivity for different values of $\frac{T_{\text{walk}}}{T_{\text{add}}}$. In this experiment we use $n = 200$, $d = 20$, $n_{\text{add}} = 20$, $T_{\text{add}} = 200$ and values $10, 20, 30$ and 50 for T_{walk}. We compute the expansion and connectivity of the graph every single round. The plot shows 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 computed for all nodes in the tree, not just the expander nodes so that we can see the time it takes for the new nodes to be added to the expander. 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 $\frac{T_{\text{walk}}}{T_{\text{add}}}$ ratio implies that nodes look for expander neighbors slowly while the graph is changing fast. This results in the graph taking a longer time to achieve better expansion and connectivity as shown in the figure.

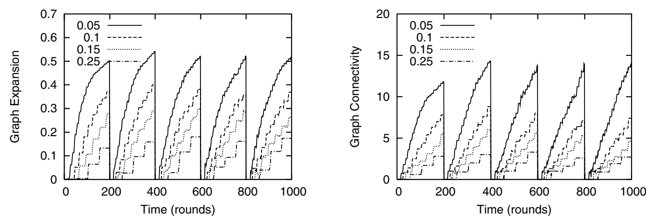

Figure 5: Expansion and Connectivity for various values of $\frac{T_{\text{walk}}}{T_{\text{add}}}$. 20 nodes are added every 200 rounds. Shows how quickly new nodes join the expander.

Figure 6 shows network behavior in the presence of node failures. Nodes run the algorithm from Section 6 to reconnect the tree after their neighbors fail so they can still run BIwalks. For this experiment we use $n = 150, d = 20, n_{\text{remove}} = 10, T_{\text{walk}} = 10$ and $T_{\text{remove}} = 300$. All 150 nodes are first added to the tree and we wait 1000 rounds for the expander to be constructed (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 node failures affect the graph expansion and connectivity slightly and the expander attempts to regain any lost fault tolerance during stable periods.

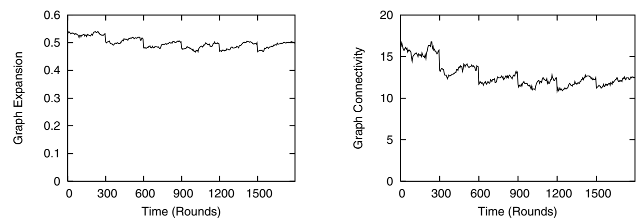

Figure 6: Expansion and Connectivity as nodes fail. 10 nodes fail every 300 rounds. Failures have a minor effect and any lost expansion and connectivity is regained in stable periods.

Figure 7 plots graph expansion and connectivity against different values of d. We use $n = 200$, $n_{\text{add}} = 20$, $T_{\text{add}} = 100$, $T_{\text{walk}} = 10$ and varied $d = 6, 10, 15, 20, 25, 29, 33$ and 38. For each value of d, we waited 1500 rounds after adding all nodes to the graph to give enough time to construct the expander and then measured expansion and connectivity. For each d, we repeated this process 30 times on different random trees and plot the average expansion and connectivity of these 30 results. As shown in the figure, expansion and connectivity increase with d. Our graphs achieve reasonable fault tolerance even for small values of d like $d = 10$.

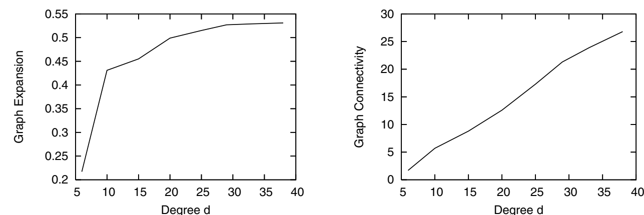

Figure 7: Expansion and Connectivity for various values of d.

Figure 8-(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 $n = 2500, d = 20, n_{\text{add}} = 10, T_{\text{add}} = 100$ and $T_{\text{walk}} = 10$. We start measuring the load from the first round

IEEE COMPUTER SOCIETY

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 25000 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 MDwalks run longer as the number of nodes increases—we use MDwalks of length $5\log(m)$, where m is the number of nodes in the expander (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 is higher than the load on other nodes only by a constant amount, even as the number of nodes increases. This constant can also be controlled using the parameter p (see lines 7–10 in Figure 2). 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 8-(b) plots the mean load per node against the level in the tree, with root at level 0. We use the same values for all the parameters as in Figure 8-(a), except n_{add} which is varied $n_{\text{add}} = 10, 15, 20, 25$. Higher n_{add} implies a more dynamic tree and thus results in more BIwalks causing a higher load on nodes close to the root. Smaller n_{add} implies a less dynamic tree resulting in more MDwalks and a better distribution of load across all nodes in the tree.

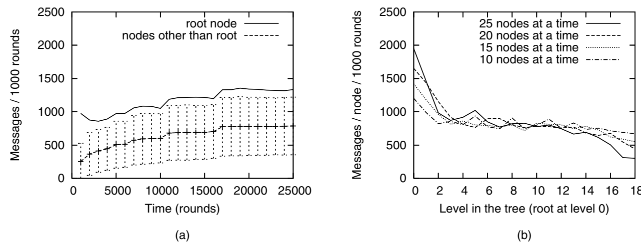

Figure 8: (a) Load on root vs mean load on other nodes. Root load is higher only by a constant amount. (b) Load at different levels is better distributed when tree is less dynamic.

8 Conclusions

We present a distributed algorithm that allows applications to exploit the acyclic and hierarchical properties of a tree without compromising the fault tolerance of the system. This is achieved by constructing an expander graph from the tree. 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 comparable connectivity and expansion. Our construction tolerates node failures using a self-healing approach. We also present an algorithm to patch the underlying tree using the fault-tolerant overlay network in case of node failures. The simulation results show that the expander graph generated has high expansion and connectivity, converges quickly after addition or removal of nodes and distributes load almost uniformly when the tree is not too dynamic.

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] D. Boggs. Internet broadcasting. Technical Report CSL-83-3, XEROX Palo Alto Research Center, 1983.

[7] 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.

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

[9] Y. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. 9th IEEE Symp. on Reliable Dist. Syst.*, pages 146–154, 1990.

[10] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proc. 5th Annual International Conference on Mobile Computing and Networks (MobiCom)*, 1999.

[11] M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. In *Proc. 12th International Symposium of Distributed Computing*, pages 119–133, 1998.

[12] A. M. Despain and A. D. Patterson. X-tree: A tree structured multi-processor computer architecture. In *Proc. 5th Annual Symposium on Computer Architecture*, 1978.

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

Proceedings of the 2005 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)
0-7695-2463-X/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

[14] O. Gabber and Z. Galil. Explicit construction of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420, 1981.

[15] L. Garcs-Erice, E. Biersack, P. Felber, K. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. In *Proc. International Conference on Parallel and Distributed Computing*, 2003.

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

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

[18] J. R. Goodman and H. S. Carlo. Hypertree: A multiprocessor interconnection topology. *IEEE Transactions on Computers*, C-30(12):923–933, 1981.

[19] J. Goshi and R. E. Ladner. Algorithms for dynamic multicast key distribution trees. In *Proc. 22nd Symposium on Principles of Distributed Computing*, July 2003.

[20] J.-M. Hlary, A. Mostefaoui, and M. Raynal. A general scheme for token- and tree-based distributed mutual exclusion algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 5(11):1185–1196, 1994.

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

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

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

[24] N. Kotilainen, M. Weber, M. Vapa, and J. Vuori. Mobile Chedar: A peer-to-peer middleware for mobile devices. In *Proc. 3rd IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW)*, 2005.

[25] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

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

[27] 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.

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

[29] H. Lu. A novel high-order tree for secure multicast key management. *IEEE Transactions on Computers*, 54(2):214–224, 2005.

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

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

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

[33] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, Nov. 1987.

[34] M. Naimi, M. Trehel, and A. Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.

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

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

[37] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, Feb. 1989.

[38] 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.

[39] K. Rothermel and C. Maihöfer. A robust and efficient mechanism for constructing multicast acknowledgement trees. In *Proc. 8th International Conference on Computer Communications and Networks ICCCN*, 1999.

[40] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

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

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

[43] SUN Microsystems. Jini technology architectural overview. `http://www.sun.com/software/jini/whitepapers/architecture.html`, Jan. 1999. White Paper.

[44] S. Wang and S. Lang. A tree-based distributed algorithm for the k-entry critical section problem. In *IEEE International Conference on Parallel and Distributed Systems*, Dec. 1994.

IEEE COMPUTER SOCIETY