Resilient Forest Routing: A Practical Approach for Single Node Failure Recovery in Sensor Networks

Paper #1569449999

***Abstract*—When nodes fail in a sensor network, it is generally accepted that there will be some transient routing failures while the routing algorithm repairs the routing state. We challenge this assumption and describe a new approach to guarantee routing correctness in the event of a single node failure using a new distributed data structure called a *resilient forest*. A resilient forest consists of several spanning trees and a set of connected local subgraphs called *resilient neighborhoods*. We illustrate our approach with *Resilient Forest Routing* (RFR), a new practical geographic routing algorithm that uses this new distributed data structure to recover from single node failures instantaneously. We show with simulations that RFR produces recovery paths that are 10% better than Bhosle, the state-of-art distributed recovery algorithm, for sparse networks and matches the performance of Bhosle for dense networks. RFR however requires only half as much storage.**

I. Introduction

When a sensor node fails, many routing algorithms take time, typically in the order of minutes, to fully repair and recompute their routing state. For some applications, this delay might be acceptable; for delay-sensitive or loss-sensitive applications, a delay of several minutes might not be acceptable. Also, when the link or node failures are only transient, it may not make sense to recompute the global routing state immediately when a node failure is detected.

In this paper, we present a new approach to guarantee packet delivery in the event of single node failures in sensor networks. Specifically, we describe a practical distributed solution to what is typically referred to as the *single node failure recovery (SNFR)* problem [1]. The new class of routing algorithms arising from our proposed methodology, which we call *resilient routing* algorithms, would allow packets to be delivered correctly even in the midst of repair, which can be done in the background. When the routing algorithm finishes repairing the routing state, the old routing state is purged. In the event of transient failures, our methodology also allows the system to initiate the global repair of routing state "lazily", by providing guarantees on packet delivery during transient outages.

Médard et al. proposed a centralized algorithm to compute a redundant tree structure to solve the SNFR problem in the context of a wired topology [2]. A key drawback of Médard's algorithm is that it assumes the network topology is *two-node-connected*, i.e. the failure of a single node will not cause a network to be partitioned. This strict requirement on the topology makes the algorithm impractical for real sensor networks. More recently, Bhosle et al. developed a distributed solution that builds a spanning tree for each destination node [3]. Since sensor network nodes generally have limited storage, it is often not practical to build one tree for each destination. One possible variation of the algorithm is to use the same spanning tree for packets to all destinations while the computation of the tree remains the same.

In our work, we build on previous work by developing a new practical distributed algorithm, called *Resilient Forest Routing (RFR)*, that makes no assumption on the network topology. In particular, the topology does not need to be two-node-connected. While our implementation is based on GDSTR [4, 5], the techniques employed could conceivably be applied to other point-to-point routing algorithms.

Our algorithm is based on two key insights: (i) in a routing algorithm that uses a spanning tree to achieve delivery guarantees [4, 3], the failure of leaf nodes does not affect delivery guarantees for the rest of the network. A large proportion of nodes in such algorithms are leaves; (ii) for the small number of remaining nodes that are not leaves, we can build local subgraphs, called *resilient neighborhoods*, that will allow us to route around the node if it fails. These are the data structures that gave rise to the name of our algorithm.

In addition to building routing data structures to achieve resilience, RFR implements two optimizations to improve routing performance. One, RFR uses a two-hop greedy forwarding strategy [5] as the basic routing algorithm in place of the typical one-hop greedy forwarding strategy, because two-hop neighbor information is readily available from the stayalive messages of its neighbors, and two-hop greedy forwarding is often sufficient to recover from many single node failures. Two, RFR exploits the broadcast nature of sensor networks to explore different parts of the network topology simultaneously when routing in the vicinity of the failed node during failure recovery.

Our simulations show that the *recovery hop stretch* of RFR is below 2 for sparse networks (average node degree 5) and approaches 1 quickly for dense networks (average node degree above 11). The recovery hop stretch is defined as the ratio of the number of hops for a route with a node failure to the minimum number of hops required to route around the failure. This compares well with Médard [2], which has a hop stretch typically above 3 for the range of network densities studied. Compared to Bhosle [3] (modified to use the same tree for all destinations), RFR achieves a recovery hop stretch that is 10% better for sparse networks and a comparable recovery hop stretch for dense networks.

Overall, RFR requires about twice as much storage as GDSTR, but only half as much storage as Bhosle. Although

RFR may duplicate the packet when it performs parallel exploration during failure recovery, the overhead is generally below 10% for sparse networks and negligible for dense networks. The maximum storage requirement for RFR for networks with up to 500 nodes is less than 1,500 bytes. This suggests that RFR is practical for deployment in existing sensor networks.

The rest of the paper is organized as follows: in Section II we list the related works; in Section III we describe in detail how to construct the data structures used by RFR for resilient routing; in Section IV we describe the routing algorithm; in Section V we evaluate the RFR algorithm and finally, we conclude our paper in Section VI.

II. Related Work

Two main classes of solutions for the SNFR problem have been proposed in the literature: (i) explicitly compute alternative paths [6, 2, 7] or (ii) build resilient data structures from which the routing algorithm can derive an alternative path [8, 3, 9]. RFR adopts the latter approach.

Routing resilience is a topic of interest in practical sensor networks. Ganesan et al. first proposed achieving routing resilience by computing disjoint and braided multipaths [6]. Their algorithm differs from ours as it computes multiple paths for a specific source and destination pair, as opposed to alternative paths for potentially any pair of nodes. While their algorithm could in theory be applied to any arbitrary pair of nodes, scalability is a concern because it is not practical to systematically enumerate and compute multiple paths between $\binom{n}{2}$ pairs of nodes for a network of size n.

Zeng et al. proposed a hyperbolic embedding algorithm for resilient routing in [10]. Paths with different homotopy types connecting the source and the destination are mapped to different images of the destination, and greedy routing to a particular image is guaranteed to find a path with the corresponding homotopy type. When failure is encountered, the node forwards the packet along a path to a different image of the destination. This approach, however, requires the network topology to be at least quasi-UDG which is hard to guarantee in practice. The computation of the embedding also incurs a huge communication overhead that is unacceptable for practical sensor networks.

Médard et al. presented a centralized solution to the SNFR problem [2] that is first algorithm for a class of algorithms called *Colored Trees* [2, 7]. This approach involves the construction of two spanning trees with a common root, such that the two paths from any node to the common root along the two trees are node-disjoint. A key assumption in this class of algorithms is that the network is two-node-connected, i.e. there does not exist any critical nodes, whose failure can partition the network. When critical nodes exist, the two colored trees constructed by these algorithms cannot span the entire network. Consequently, packets may not be delivered correctly even if the failed node is not a critical node. In contrast, RFR is designed to handle critical nodes in the topology and can provide delivery guarantees for all non-critical node failures, even in topologies containing critical nodes.

There are some approaches that address the single node failure problem and the related link failure problem based on interface-specific forwarding [8]. Instead of explicitly computing alternative paths, these algorithms build interfaces so that node failures can be inferred if a packet is received via a special interface instead of a normal interface. The node then forwards the packet through the special interface which guarantees that the failure will not be encountered. However, these are centralized algorithms which assume each node has global knowledge of the network topology.

Bhosle and Gonzalez proposed a distributed algorithm to handle single node failures based on a shortest path tree [3]. Their algorithm pre-computes the alternative path between any child of the failed node and the parent of the failed node. However, their routing protocol [3] assumes that the destination of the packet is the root of the shortest path tree. While this is convenient in that the algorithm is only required to forward the packet up the tree, this assumption requires the algorithm to rebuild the shortest path tree and recompute all alternative paths whenever the destination of the packet is not the root of the current shortest path tree. Alternatively, the network can construct n spanning trees such that each node in the network will be a root of its own the trees. However, in the context of memory-constrained sensor devices, it is not practical and scalable to maintain $O(n)$ state.

The *Resilient Routing Layer* (RRL) approach [9] is similar to the leaf-covering mechanism in RFR. A resilient routing layer is a connected sub-graph containing all the vertices but only selected edges from the original graph. These algorithms attempt to create several resilient layers so that each node is of degree 1 (or what is equivalent to a leaf in RFR) in at least one layer. However, this approach is not sufficient to ensure correctness with arbitrary topologies with finite storage. The reason is that the number of layers needed to cover an arbitrary network topology is unbounded.

It turns out that existing state-of-art point-to-point routing algorithms for sensor networks [5], like S4 [11] and GDSTR [4], already possess some degree of resilience to node failures. GDSTR employs greedy forwarding which can route around failed nodes in dense networks because an alternative node that provides some progress to the destination can easily be found. S4 implements a mechanism called *distance guided local failure recovery* (DLF). When the failed node is encountered, a failure recovery request is broadcast to all the neighbors. The packet is then forwarded to the neighbor that responds first. Both these approaches improve resilience but do not provide any guarantees of packet delivery in the event of an arbitrary single node failure.

III. Building the Resilient Forest

The RFR algorithm consists of two key components: (i) an algorithm to build the required resilient distributed data structure (stored in the nodes), and (ii) a routing algorithm

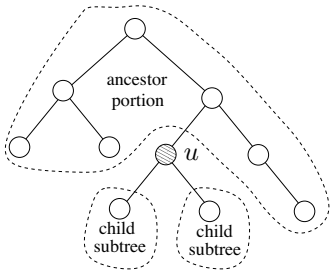

Fig. 1. Network components relative to a critical node.

that leverages on the resilient distributed data structure to achieve failure resilience. In this section, we describe the former algorithm. The RFR resilient forest consists of two *resilient spanning trees* and *resilient neighborhoods* for each non-critical node that is not a leaf in either of the resilient spanning trees.

RFR employs GDSTR as the underlying routing protocol when there are no node failures. Like GDSTR, RFR employs two distributed spanning trees that are augmented hull trees. To achieve good routing performance, the two trees should be rooted at opposite ends of a network [4]. This is achieved by electing the two nodes with maximal and minimal x-coordinates as the two roots. This election can be done with a simple flooding protocol: each node first considers itself to be the root of both trees. Periodically, the nodes will broadcast their views of the two roots. Whenever a node learns about a node that has a larger (or smaller) x-coordinate than its current views of the two roots, it updates its view. In this way, the network will quickly converge to the same view on the two roots.

One tree is constructed exactly like GDSTR by having each node pick a parent that minimizes its hop count to the root. This tree is augmented with connectivity information and is used in the critical node detection algorithm described in Section III-A. The second tree is constructed after the first tree has stabilized because as described in Section III-B, its construction depends on the outcome of the critical node detection algorithm.

A. Identification of Critical Nodes

A *critical node* is a node whose removal will partition a network into two or more separate components. We first identify the critical nodes in the network since it is not possible to find alternate paths to route around them for all source and destinations pairs if they should fail.

Our algorithm is an adaptation of the depth-first search (DFS) algorithm [12] to determine critical nodes. As illustrated in Fig. 1, we consider the shaded node u in a spanning tree of the network. u divides the network into a number of child subtrees and an ancestor portion. To decide whether u is a critical node, it suffices to check whether the ancestor portion and the child subtrees remain connected to each other upon the removal of u. When u is removed, a child subtree can be connected to the ancestor portion in two ways:

- A child subtree is *directly* connected to the ancestor portion if there is an edge (x, y) such that x is in the ancestor portion and y is in the child sub-tree; or
- A child subtree is *indirectly* connected to the ancestor portion if it is *not directed connected*, but is connected to another child subtree that is either connected directly or indirectly to the ancestor portion.

Algorithm 1: isCritical(u)

1: Identify the set containing all the children of u whose subtree is *directly* connected to the ancestor portion.
2: Compute the connectivity between different child subtrees.
3: Check whether each child not in the set is connected through a path to a child in the set. If so, then u is not critical; otherwise u is a critical node.

It is clear that a node is a critical node, iff there exists a child subtree that is not connected either directly or indirectly to the ancestor portion. With this definition, we can identify critical nodes with Algorithm 1.

The last step in Algorithm 1 can be easily implemented by breadth-first search (BFS). The main challenge is to implement the first two steps efficiently in a distributed fashion. We implement these two steps with a simple tree-based labelling and aggregation algorithm using one of the RFR resilient trees (since any arbitrary spanning tree will suffice).

Let x be a node in an arbitrary child subtree. Note that in the first step, we need to determine if x has a neighbor in the ancestor portion, and in the second step, we need to determine if x has a neighbor in another child subtree. In other words, if a node u wants to know whether it is a critical node, it requires the following information from every node x in its child subtrees:

- **Subtree resilience**: Whether there is an edge (x, y) where y is an ancestor of u.
- **Subtree connectivity**: For every child subtree $\mathcal{T}$ of u, whether there is an edge (x, z) where z is in $\mathcal{T}$.

Moreover, we note that if x is in a child subtree of u, then x must be in a child subtree of v for all ancestors v of u. Therefore, from the viewpoint of x, it needs to provide the above information to u and all the ancestors of u. We observe that if nodes are appropriately labelled and each node broadcasts its label then each node can answer the above questions from the labels of the nodes in its one-hop neighborhood. Therefore, instead of repeatedly looking at the neighborhood of x to collect information for different ancestors, what we do is to *explore the neighborhood of x only once and collect information for all its ancestors*.

It remains for us to describe the method that allows x to determine its ancestor nodes in the tree: this can be achieved by assigning a unique label to each node in the tree. The labels are assigned recursively as follows: the root is labeled

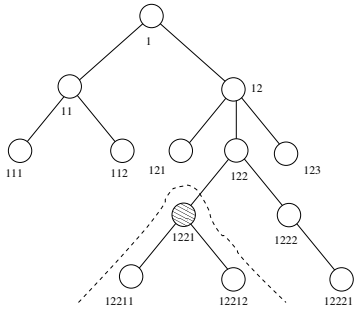

Fig. 2. Example of connectivity information aggregation one hop from the leaves.

TABLE I
CONNECTIVITY MATRIX FORWARDED BY NODE 12212 TO PARENT.

Final recipient	*Subtree resilience*	*Subtree connectivity*	
1221	¬1221	12211*	-
122	¬122	1222*	-
12	¬12	121*	123*
1	-	11*	-

"1"; if a node is labeled k, then its i-th child will use the label generated by appending i to the end of k. With this labelling, we observe that if a node v has a label extended from that of u, then it is in the sub-tree of u and therefore in the sub-tree of each ancestor of u. An example of such a labelling is shown in Fig. 2.

The important property of this labelling is that, *u is an ancestor of x if and only if the label of x is an extension of the label of u*. Therefore, by looking at its own label, a node can easily determine the label of each of its ancestors. Therefore, any node x can easily determine, for any ancestor u of x, the label of the child v of u such that x is in the subtree of v. Then, x can answer the first question by checking whether it has a neighbor y whose label does not extend the label of u, and the second question by checking whether it has a neighbor z whose label extends the label of u but does not extend the label of v. Moreover, since x knows the labels of all its ancestors, it can answer the same questions for its ancestors from its one-hop neighborhood. This information is forwarded to its parent.

We use a boolean matrix to store the connectivity information forwarded by x. We call this matrix a *connectivity matrix*. The connectivity matrix is structured such that information provided to different ancestors are stored in different rows, and thus each ancestor, when receiving the matrix, will only need to use one row of the matrix. The connectivity matrix for node 12212 in Fig. 2 is shown in Table I. The boolean value at each entry shows whether node 12212 is connected to the node whose label matches the expression specified at the entry. If α is a label, we say that a label β matches $\alpha*$ if β is an extension of α, and matches $\neg\alpha$ if β is not an extension of α.

We use a tree aggregation approach to reduce message overhead. Instead of having each node send its connectivity matrix to each of its ancestors, the connectivity matrix is propagated and aggregated up the tree. A node will aggregate the connectivity matrices received from its child nodes by performing a boolean *OR* operation between corresponding elements of the matrices and its own connectivity matrix. With the aggregation operation, the interpretation of the aggregated matrix is slightly different from that of a vanilla connectivity matrix: while an element in the connectivity matrix of x indicates whether x is connected to a node y whose label satisfies certain conditions, an element in the aggregated matrix indicates whether the subtree of x is connected to the subtree of y whose label satisfies the same conditions. A node can then obtain the subtree resilience information and the subtree connectivity information of a child node by checking appropriate entry in the aggregated matrix forwarded by the child.

B. Resilient Spanning Trees

The idea of using spanning trees for resilient routing comes from the observation that a leaf node in a spanning tree will not forward transit traffic. Therefore, if a leaf node fails, a packet may still be delivered if neither the source nor the destination is the failed node. Routing resilience can be achieved if we can build trees in a way such that each non-critical node is a leaf in at least one tree. This suggests that we should construct trees in such a way that the number of nodes that are leaves in at least one tree is maximized.

Since we already have one spanning tree that was constructed to identify the critical nodes in the network, it remains for us to construct the second spanning tree in such a way that it covers as many nodes which are not covered as leaves in the first tree as possible. We observe that each non-root node in the second tree has some amount of flexibility in choosing its parent node. Intuitively, nodes should avoid choosing nodes that are not leaves in the first tree as parents, since doing so would render them non-leaves in both trees; however a node could do better to avoid not just a non-leaf parent, but the entire branch to the root that contains some non-leaf ancestor.

In particular, when we build the second tree, a node will prefer a parent whose path to the root contains only critical nodes (which are impossible to cover as leaves) or nodes that have already been covered as leaves in the first tree. When no such parent is available, a node prefers a parent whose least uncovered ancestor has a smaller id. This information is propagated down the tree incrementally.

C. Base Routing Protocol Without Node Failures (GDSTR)

While RFR could in principle be applied to any other point-to-point routing algorithm, we choose GDSTR [4] as the base routing algorithm when there are no node failures for the following reasons: (i) GDSTR achieves good routing performance with affordable cost in sensor networks [5]; (ii) GDSTR is fully distributed; and (iii) GDSTR also employs global spanning trees which can be naturally augmented for the purpose of resilient routing as described in Section III-A and III-B.

GDSTR forwards packets using simple greedy forwarding whenever possible. It switches to a forwarding mode based on a *hull tree* only to route packets around "voids," and

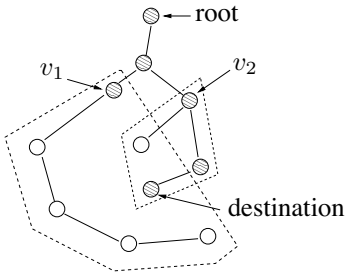

Fig. 3. Overview of GDSTR tree traversal (shaded nodes belong to the routing subtree).

when a packet ends up in a local minimum. A hull tree is a spanning tree that aggregates the locations covered by subtrees using convex hulls. GDSTR uses the convex hull information to decide the direction that is most likely to make progress towards a given geographic destination. It switches back to greedy forwarding as soon as it is feasible to do so.

The key insight in GDSTR is that the hulls of the nodes uniquely determine a *routing subtree* containing the root of the hull tree within which a destination must be located, if it exists. In particular, the routing subtree is defined as the set of nodes which have hulls that contain the destination point. If a packet is not deliverable, the routing subtree will be a null tree. In the example shown in Fig. 3, both v_1 and v_2 are on the routing subtree since their hulls contain the destination.

Tree forwarding involves first routing to the *routing subtree*. Basically, if a node is not on the routing subtree, it forwards the packet up to its parent. Eventually, the packet will either reach a node on the routing subtree or the root (which is guaranteed to be on the routing subtree if a packet is deliverable). Once the packet reaches a node on the routing subtree (called the *anchor*), tree traversal is simply a depth-first search on the routing subtree that explores the children first. If a packet returns to the anchor node from the parent without having been delivered, we conclude that the packet is not deliverable. GDSTR will switch back to greedy forwarding at the first opportunity that it finds a neighbor closer to the destination than the local minimum that initiated the tree traversal.

D. Resilient Neighborhood

A naive solution to the SNFR problem would be to build sufficient resilient trees to cover all the non-critical nodes as leaves in some tree. Unfortunately, according to Lemma 1, it is impossible to bound the number of trees that is needed to cover all the leaves in an arbitrary topology.

> **Lemma 1:** *There does not exist an integer N such that, given any connected network, there is a way to build N trees where each non-critical node is a leaf in at least one tree.*
>
> *Proof:* Suppose such an N exists. Consider a circle of $2N + 1$ nodes. Each spanning tree of the circle has exactly 2 leaves. Therefore, with N spanning trees one can cover at most $2N$ nodes as leaves, contradicting our assumption that N spanning trees are always sufficient. ■

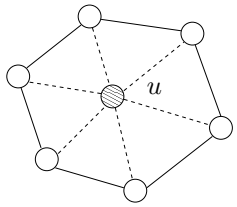

Fig. 4. Example of a connected resilient neighborhood.

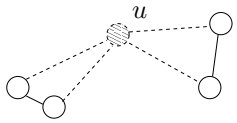

Fig. 5. Example of a resilient neighborhood consisting of multiple components.

It turns out that typically after building two resilient trees, only a small proportion of the nodes are not covered as leaves in either tree. This suggests that if we have a way to route around these remaining nodes if they should fail, the SNFR problem is solved.

Next, we note that in the event of a single node failure, a packet is deliverable if the failed node is neither the source nor the destination and any two neighbors of the failed node remains connected upon the removal of the failed node from the network topology. This observation suggests that to guarantee packet delivery with the failure of a non-critical node, it suffices to find paths connecting *any two neighbors* of the failed node, instead of paths connecting *any two nodes* in the network topology. The notion of a resilient neighborhood arises naturally from this insight.

> **Definition 1:** A set S of nodes is a resilient neighborhood of u if $u \notin S$ and for any two neighbors x, y of u, there is a path connecting x and y which only consists of nodes in S.

According to Definition 1, any non-critical node u has a resilient neighborhood, which is simply the set of all nodes minus u itself. However, for most nodes there exists a much smaller set of nodes that forms its resilient neighborhood. For the node u in Fig. 4, the minimal resilient neighborhood is simply the set of all its neighbors. It is also possible for the minimal resilient neighborhood of a node to consist of nodes beyond its immediate neighbors. For example, in the case of the node u shown in Fig. 5, where the immediate neighbors consists of two components, the resilient neighborhood must contain some non-neighboring nodes.

Computation of Resilient Neighborhood. To compute its resilient neighborhood, a non-critical node u that is not covered as a leaf in the two resilient spanning trees, constructs a two-hop local spanning tree rooted at itself in a distributed way. A simple broadcast-based algorithm is sufficient because the structure of the tree is not important. The nodes in the trees are labelled in a manner similar to that of the critical node detection algorithm described in Section III-A. Once the tree is constructed and labelled, the leaves of this tree will propagate connectivity information up the tree. This connectivity

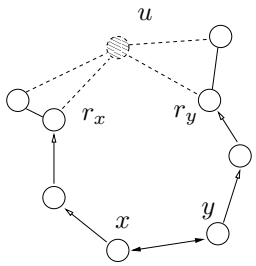

Fig. 6. Resilient neighborhood computation.

information is sufficient for u to compute the paths required to connect its components. It is possible that the required paths cannot be found within a two-hop neighborhood. If so, u will repeat the process and increase the maximum depth of the local tree by one. We know that such paths must exist because node u is not a critical node and so u will iteratively increase the depth of this search until the paths required to connect all its one-hop neighbors are found.

In the example network shown in Fig. 6, the node u constructs a 4-hop local tree. A node x joins the tree whose ancestor is r_x, and x observes that one of its neighbors y is in another child subtree of u with ancestor r_y. Since r_x and r_y, which are both neighbors of u, are in different components in the subgraph consisting of the neighbors of u, x has witnessed a path connecting these two components, which is simply the tree path of x to its ancestor r_x and the tree path of y to its ancestor r_y joined by the bridging edge (x, y). After witnessing such a path, x will report this information up to its ancestor r_x, which then forward it to u using a connectivity matrix similar to that described in Section III-A.

Once u computes its resilient neighborhood, this information is broadcast to all the nodes in the neighborhood using the local tree. The resilient neighborhood is a connected subgraph and each node in the neighborhood only needs to store its direct neighbors in this subgraph and not the full subgraph. The reason why this approach is scalable is because we only need to compute these neighborhoods for small proportion of the nodes in a typical network and these resilient neighborhoods tend to be small.

E. Incremental Changes

We assume that the network remains stable until the resilient spanning trees and resilient neighborhoods are built. However the RFR algorithm is also able to take care of node churns locally as follows:

Resilient Spanning Tree. When a new node joins the network, we attach the new node as a child to one of its neighbors, preferably in a tree where the neighbor is not a leaf, so that the new node is always a leaf and all existing leaves remain leaves. When some node leaves the network, all the nodes in its subtree will clear their state and attempt to re-attach themselves to the tree.

Critical Node Detection. When a new node joins a network, all existing non-critical nodes will clearly remain non-critical. However, a new node may improve the connectivity of the network and cause some critical nodes to become non-critical. Therefore, we need to add this new node to the critical node detection tree and let the new node compute and send the updated aggregated matrix up to its parent. The parent will re-check whether it is a critical node using the updated information and send its updated aggregated matrix to its parent. This process ends until all nodes along the path from the new node to the root has updated their connectivity matrices with the knowledge of the new node, which incurs d messages in total where d is the depth of the new node in the tree. If some node leaves the network, we wait for the spanning tree used for critical node detection to re-stabilize. All the nodes in the tree that have a new label or has a one-hop neighbor with an updated label arising from the change will need to update its connectivity matrix and send the updated aggregated matrix up to its parent, which in turn updates its own connectivity matrix and propagates the change up the tree. New critical nodes will be detected with the updated connectivity matrix information.

Resilient Neighborhood. When a new node joins the network, we compute the resilient neighborhoods of the new node and of those nodes which cease to be critical. When some node leaves the network, we re-compute the resilient neighborhoods whose local trees are disconnected by the removal of the node.

IV. Routing Algorithm

Resilient Forest Routing performs exactly like GDSTR when there are no node failures. Packets are forwarded greedily to the next hop neighbor that is closest in geographic distance to the destination. When a packet reaches a local minimum, the packet is routed on the spanning tree which has a root that is nearer to the destination [4]. Resilient Forest Recovery is activated when the basic GDSTR routing algorithm attempts to forward a packet to a failed node. As described in Algorithm 2, the Resilient Forest Recovery algorithm consists of three modes: (i) two-hop greedy forwarding, (ii) fixed tree traversal, and (iii) forest exploration. For clarity, the associated subroutines are listed separately as Algorithms 2(a), 2(b) and 2(c).

Two-hop Greedy Forwarding. In two-hop greedy forwarding, a node examines the two-hop neighborhood to determine the next hop. While this requires a node to store the neighbors of each of its own neighbor, it does not impose much additional cost, since the required information is already contained in the periodic stayalive messages and a two-hop neighborhood is typically relatively small. While traditional one-hop greedy forwarding may get stuck at node x if the chosen next hop u is a failed node, two-hop greedy forwarding has the potential of finding another intermediate neighbor v other than u such that the packet could be forwarded first to v and then to some node y in the neighborhood of v, which is closer to the ultimate destination than x.

Fixed Tree Traversal. If two-hop greedy forwarding is not able to recover from the node failure, we will attempt to use the resilient forest for recovery. We observe that if the failed node is a leaf in one of the two spanning trees, we can recover by simply forwarding the packet on the tree where the failed node is a leaf. We call this *fixed tree traversal* because for

Algorithm 2: Resilient Forest Recovery

```
if a failed node is encountered then
    TwoHopGreedyForwarding()
    if two-hop greedy forwarding fails then
        if dead node is leaf in some tree 𝒯 then
            FixedTreeTraversal() using 𝒯
        else
            ForestExploration()
```

all intents and purposes it is identical to the tree traversal algorithm for GDSTR, except for two minor differences: (i) the packet is not allowed to switch back to greedy forwarding and (ii) it cannot change the underlying spanning tree being used.

Forest Exploration. Recall that given a spanning tree $\mathcal{T}$ of the network topology, the failure of u will divide $\mathcal{T}$ into an ancestor component and several child subtree components. The destination node will be contained in one of these components if u is not the destination. Clearly, any tree neighbor of u in $\mathcal{T}$ is a one-hop neighbor of u and the resilient neighborhood of u connects all the components.

The node that detects the node failure must be a neighbor of the failed node u, and if two-hop forwarding cannot route around u and the failed node u is not a leaf in one of the two spanning trees, we flood the packet to the entire resilient neighborhood of u. This ensures that every component of u will receive the packet. When the packet arrives at a node that is a tree neighbor of u in $\mathcal{T}$, we forward a copy of the packet to explore the associated component in $\mathcal{T}$ using fixed tree traversal (Algorithm 2(b)). The resilient neighborhood of u is typically either a tree or a ring, so few duplicate packets are generated in the flooding.

Some duplicate packets are created in the parallel exploration of the neighborhood components. However, during fixed tree traversal, the only way to escape a component is to forward the packet to the failed node u, and if a packet attempts to visit the failed node after all the rest tree neighbors in the portion have been visited, the neighboring node of u will realize that the component does not contain the destination and drop the duplicate packet.

We use fixed tree traversal during forest exploration because not doing so might cause the packet to be forwarded from one component to another along an edge that is not in $\mathcal{T}$. If this happens, the packet might leave a component without exploring it thoroughly, and if so, we will not be able to detect and drop duplicate packets reliably. If this happens, the destination may also receive duplicate packets.

V. Performance Evaluation

We evaluated the performance of RFR with a high-level event-driven simulator *netsim2* [13]. Wireless losses are not simulated since our goal is to evaluate the algorithmic behavior of RFR and compare it with other algorithms. We evaluated the algorithms with connected random unit disk graph (UDG)

Algorithm 2(a): Two-Hop Greedy Forwarding

```
Suppose n is the closest to the destination among all neighbors within 2 hops
if n is closer to the destination than the current node then
    if n is a one-hop neighbor then
        forward packet to n
    else
        forward packet to any one-hop neighbor connected to n
else
    two-hop greedy forwarding fails
```

Algorithm 2(b): Fixed Tree Traversal

```
if the packet is at the anchor node and is forwarded from the parent then
    packet not deliverable
if there exists an unvisited child c (not the failed node) whose hull contains destination then
    forward packet to c
else if current node is not root then
    forward packet to the parent
else
    packet not deliverable
```

networks with average node degrees between 5 and 16. Denser networks are not interesting as we will show that two-hop greedy forwarding is almost always sufficient to recover from node failures in such networks.

While the simulator supports obstacles, we use UDGs for simplicity, since RFR is based on GDSTR, which is oblivious to location errors and crossed links unlike earlier geographic routing algorithms [14], and adding obstacles would only serve to reduce the effective node degrees for the topologies and make them equivalent to those of lower density without adding too much value. The random networks are generated by assuming unit radio range and randomly scattering 500 nodes over a $n \times n$ square and removing all the nodes that are not connected to the maximally connected subgraph. The error bars in our results reflect the 95% confidence intervals.

A. Characterizing Resilient Neighborhoods

A key insight in our approach is that the resilient spanning trees are able to cover a significant proportion of the nodes

Algorithm 2(c): Forest Exploration

```
flood the packet so that each neighbour u of the failed node receives a copy
if the hull of u contains destination then
    duplicate the packet and send a copy to explore the corresponding component using Fixed Tree Traversal
```

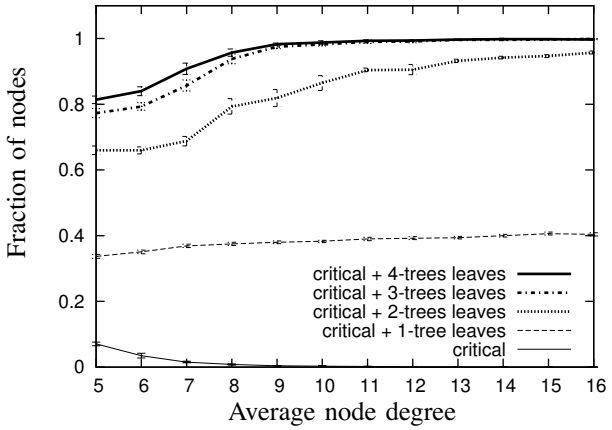

Fig. 7. Coverage of nodes as leaves by resilient spanning trees for 500-node random networks.

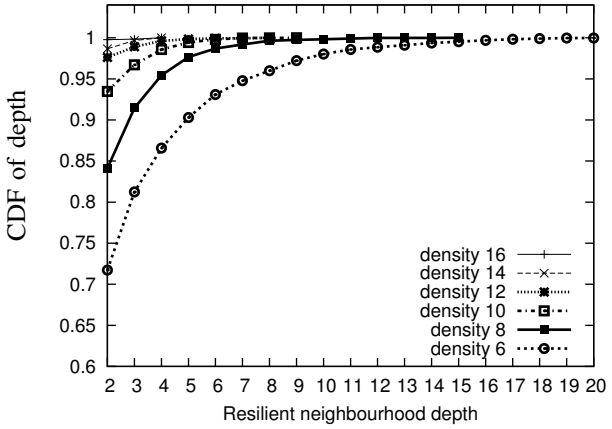

Fig. 8. Cumulative distribution of resilient neighborhood depth.

cheaply, so we only have to compute resilient neighborhoods for a small number of nodes and do not need invoke forest exploration often. In Fig. 7, we plot the proportion of nodes covered by leaves, which corresponds to the lower bound of the proportion of nodes whose failure will *definitely not* invoke the forest exploration mode for any packet. We see from Fig. 7 that critical nodes are more common in sparse networks and resilient trees are less effective at covering the nodes. For sparse networks (average node degree 5), two resilient trees can only cover about 70% of the nodes, while for dense networks (average node degree 16), leaves can cover more than 90% of the nodes. As shown, the use of three or more resilient trees is only able to improve leaf coverage marginally. RFR achieves a good tradeoff between performance and storage by using only two trees.

In Fig. 8, we plot the cumulative distribution of the resilient neighborhood depth for all the non-critical nodes that are not covered by leaves. By depth, we refer to the maximum number of hops that a resilient neighborhood can extend from the associated node. The graph shows that the resilient neighborhoods are relatively shallow, with the majority of them being at most 2 hops deep. This explains why storage requirements for the resilient neighborhoods are relatively low on average.

B. Node Failure Recovery Success Rate

As expected, RFR is able to achieve 100% packet delivery when a packet encounters a single failed node. The success rates of other algorithms are plotted in Fig. 9 for comparison. To understand the contribution of each resilient mechanism, we plot the success rate of GDSTR, GDSTR-2H (GDSTR with two-hop greedy forwarding) and GDSTR-2H with leaf covering (equivalent to RFR without resilient neighborhoods). Our results suggest that for RFR, two-hop greedy forwarding contributes more to the recovery process for dense networks, the resilient spanning trees contributes some amount for sparse networks and the resilient neighborhoods make up most of the difference for extremely sparse networks.

To obtain these results, we randomly generated pairs of source and destination nodes, routed packets between them and recorded the paths taken by each packet. Since the evaluated algorithms choose their next hop node deterministically, the collection of paths provides us with sufficient information to find routes that will pass through specific nodes. Next, we randomly picked a non-critical code in the topology to fail and, by inspecting the set of paths we computed earlier, we generated packets that would be routed through the failed node in the ordinary (non-failure) case. This allows us to measure the success rate of each algorithm in recovering from a single node failure. We repeated the experiment for 100 randomly-generated 500-node networks by routing about x packets per experiment, and choosing a random non-critical node to fail each time.

The success rates of GDSTR, GDSTR-2H and GDSTR-2H with leaf-covering improves significantly with increasing network density and is close to 1 at high density. In contrast, the success rate of S4 improves slowly and it does not converge to 1 at high densities. This is because S4 is extremely resilient only if the node failure is within a cluster containing the destination. If the node failure occurs outside the cluster containing the destination, its distance guided local failure recovery mechanism is not always able to route around the failed node.

To understand how the different forwarding modes contribute to resilience in RFR, we plot in Fig. 10 the cumulative distribution of the modes activated during failure recovery. It is clear that resilient spanning trees is able to recover from a node failure some 60% of the time, which is consistent with the proportion of leaf nodes shown in Fig. 7. In networks with high densities (> 10), the inexpensive partial recovery mechanisms like greedy forwarding and GDSTR tree traversal are already able to guarantee delivery over 90% of the time.

C. Recovery Routing Performance

In this section, we compare the efficiency of the recovery path found by RFR for a single node failure to that of other SNFR algorithms like Bhosle [1] and Médard [2].

Although Médard's algorithm has different variations, their differences lie mainly in the computation of the colored trees. None of the variations guarantee more than the property that the two spanning trees constructed will have the same root

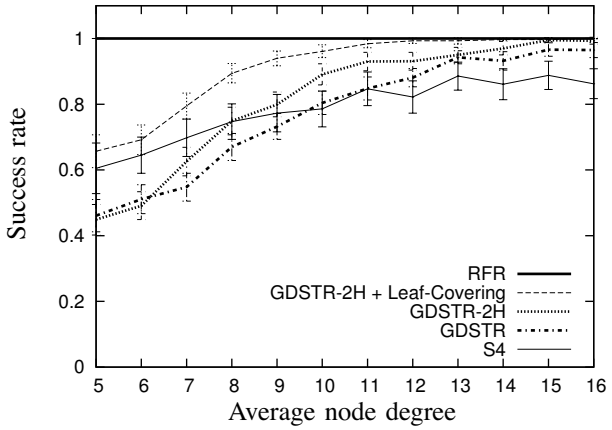

Fig. 9. Plot of packet delivery success rates in the event of single node failures for various routing algorithms.

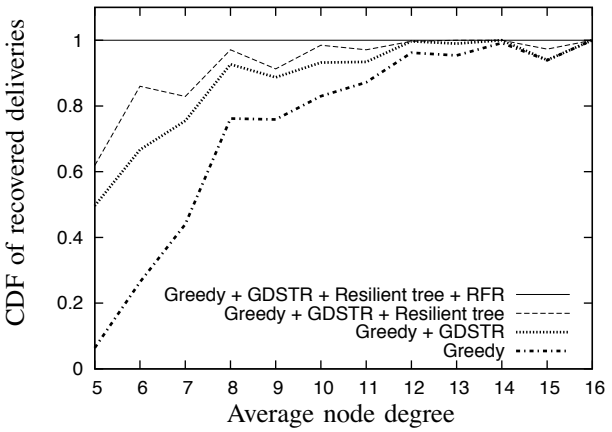

Fig. 10. Plot of CDF of recovered deliveries contributed by various mechanisms of RFR.

and that every node has a disjoint path to the root. Therefore, to measure the quality of the recovery paths, it suffices to measure the performance of Médard's algorithm. Since Médard did not specify a routing protocol when a node fails, we implemented a variant of GDSTR that invokes Médard's recovery mechanism when a node failure is encountered. We call the resulting algorithm Médard/GDSTR. Because Médard is a centralized algorithm, it functions like an oracle in our implementation.

In their original algorithm, Bhosle et al. assumed that the destination is the root of a global spanning tree [1]. Since it is not practical to build a tree for each node because of storage considerations in a sensor network, we instead implemented a variant of Bhosle's recovery mechanism that uses one spanning tree for packets to all destinations over GDSTR. We call the resulting algorithm Bhosle/GDSTR.

Our goal is not so much to present RFR as an ideal routing algorithm, but to compare the effectiveness of the resilient forest approach to Bhosle [1] and Médard [2]. This is why we compare RFR to Médard and Bhosle with GDSTR instead of a different base routing algorithm.

We see from the results in Fig. 11 that Médard/GDSTR

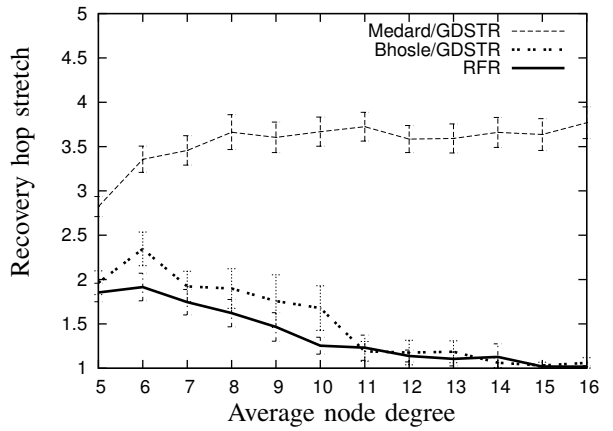

Fig. 11. Comparison of the hop stretch for recovery paths.

has the highest recovery hop stretch hovering around 3 at all densities. The recovery hop stretch is defined as the ratio of the number of hops for the path found by an algorithm to route around a node failure to the number of hops for the shortest path required to do the same. A recovery hop stretch of one implies that an algorithm finds the optimal (shortest) recovery path. The hop stretch for RFR is lower than Bhosle/GDSTR by about 10% to 20% for sparse networks and approximately the same for dense networks. RFR performs better than Bhosle/GDSTR because it is able to find shorter recovery paths in parallel using the resilient neighborhood (which is a local tree structure) instead of a global spanning tree.

D. Overhead

It remains for us to quantify the costs associated with RFR. For wireless devices with limited storage like sensor motes, the maximum node storage cost is of utmost concern [5]. In Fig. 12, we compare the storage cost of RFR to Bhosle/GDSTR, GDSTR and S4 for networks from 100 to 1,000 nodes with average node degree 10. The high storage cost of Bhosle/GDSTR algorithm is due to the fact that Bhosle uses a global spanning tree for recovery. Therefore, a node occasionally has to store an extremely long recovery path to each of its grandchildren in order to route the packet down to them in case one of its children fails. Overall RFR requires about twice as much storage as GDSTR, but only half as much storage as Bhosle. Fig. 12 also seems to suggest that the maximum storage requirement increases relatively slowly as the system scales up in size. While not shown here because of space constraints, we note that the graphs for networks of other densities are similar to Fig. 12.

The stabilization overhead is plotted in Fig. 13 for 500-node networks in terms of the number of stabilization messages sent per node. The stabilization overhead of RFR starts at around 100 messages per node for low density networks and gradually decreases and stabilizes around 50 messages per node as network density increases. The decrease in overhead is due to two reasons: (i) in dense networks, a resilient neighborhood generally has fewer components than sparse networks; and (ii)

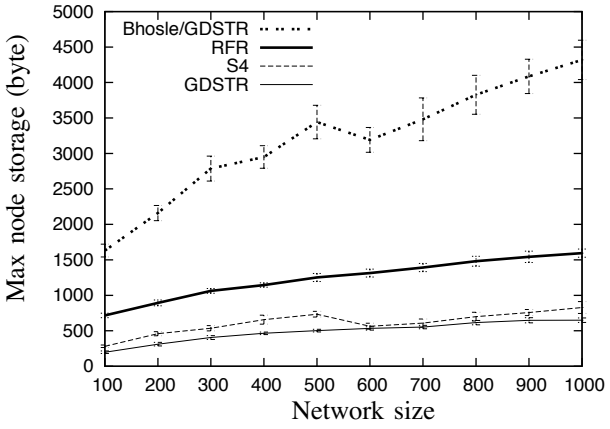

Fig. 12. Plot of maximum node storage required for networks with average node degree 10.

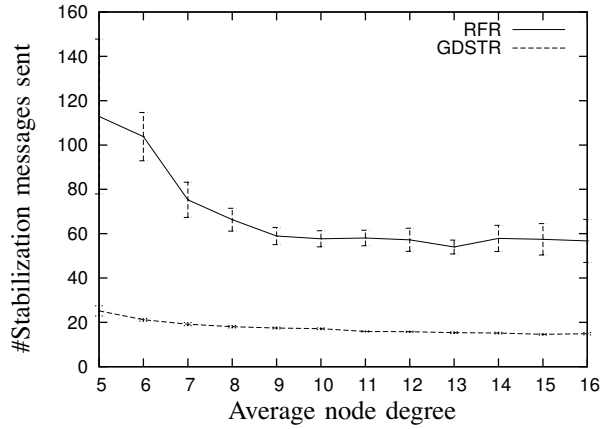

Fig. 13. Plot of stabilization overhead against network density.

the resilient neighborhood depth is generally smaller in dense networks as shown in Fig. 8.

Since RFR generates duplicate packets during forest exploration to improve performance, we measured the additional transmission overhead arising from the duplicate packets that eventually get dropped. We define the *parallel exploration overhead* of a packet to be the ratio of the sum of the additional hop counts of the dropped duplicate packets over the hop counts of the copy of the packet that eventually reaches the destination. This overhead is plotted in Fig. 14. We see that the overhead is less than 10% for sparse networks and negligible for dense networks. We can reduce this overhead over time with caching at the cost of additional storage at the nodes.

VI. Conclusion

In this paper, we present a new approach for single node failure recovery that uses a new distributed data structure called a *resilient forest*. We describe and evaluate Resilient Forest Routing and show that it is able to achieve good path recovery hop stretch comparable to Bhosle, the state-of-art distributed recovery algorithm, with less than half its storage requirement. Since the maximum storage requirement for RFR for networks with up to 1,000 nodes is less than 1,600 bytes,

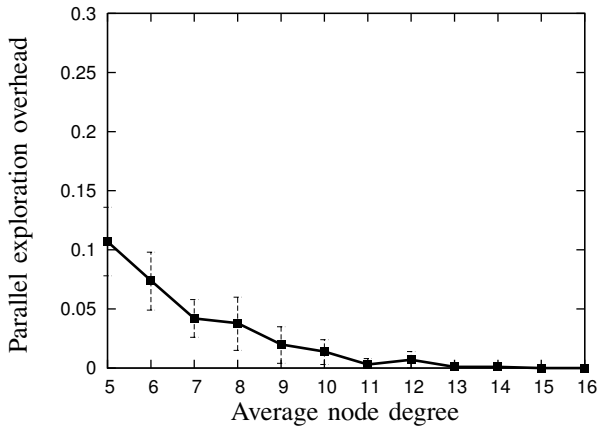

Fig. 14. Parallel exploration overhead of RFR.

RFR is suitable for practical deployment in existing sensor networks. As future work, we plan to implement RFR in TinyOS and evaluate its performance on a real sensor testbed.

References

[1] A. M. Bhosle and T. F. Gonzalez, "Distributed algorithms for computing alternate paths avoiding failed nodes and links," *CoRR*, vol. abs/0811.1301, 2008.

[2] M. Médard, S. G. Finn, and R. A. Barry, "Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs," *IEEE/ACM Transactions on Networking*, vol. 7, no. 5, pp. 641–652, 1999.

[3] A. M. Bhosle and T. F. Gonzalez, "Efficient algorithms and routing protocols for handling transient single node failures," *CoRR*, vol. abs/0810.3438, 2008.

[4] B. Leong, B. Liskov, and R. Morris, "Geographic routing without planarization," in *Proceedings of NSDI 2006*, May 2006.

[5] J. Zhou, Y. Chen, B. Leong, and P. Sundaramoorthy, "Practical 3D geographic routing for wireless sensor networks," in *Proceedings of SenSys'10*, November 2010.

[6] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin, "Highly-resilient, energy-efficient multipath routing in wireless sensor networks," in *Proceedings of MobiHoc '01*. New York, NY, USA: ACM, 2001, pp. 251–254.

[7] S. Ramasubramanian, H. Krishnamoorthy, and M. Krunz, "Disjoint multipath routing using colored trees," *Computer Networks*, vol. 51, no. 8, pp. 2163–2180, 2007.

[8] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah, "Fast local rerouting for handling transient link failures," *IEEE/ACM Transactions on Networking*, vol. 15, no. 2, pp. 359–372, 2007.

[9] T. Čičić, A. F. Hansen, S. Gjessing1, and O. Lysne, "Applicability of resilient routing layers for k-fault network recovery," in *Proceedings of ICN '05*, 2005.

[10] W. Zeng, R. Sarkar, F. Luo, X. Gu, and J. Gao, "Resilient routing for sensor networks using hyperbolic embedding of universal covering space," in *INFOCOM, 2010 Proceedings IEEE*, march 2010, pp. 1 –9.

[11] Y. Mao, F. Wang, L. Qiu, S. S. Lam, and J. M. Smith, "S4: Small state and small stretch routing protocol for large wireless sensor networks," in *In Proceedings of NSDI 2007*, 2007.

[12] R. Tarjan, "Depth first search and linear graph algorithms," in *SIAM J. Computing*, 1972, pp. 1, 2, 146–160.

[13] B. Leong, "Geographic routing network simulator," 2004, http://groups.google.com/group/netsim-users.

[14] K. Seada, A. Helmy, and R. Govindan, "On the effect of localization errors on geographic face routing in sensor networks," in *Proceedings of IPSN'04*, 2004, pp. 71–80.