CREATION AND MAINTENANCE OF A COMMUNICATION TREE IN WIRELESS SENSOR NETWORKS

A Dissertation

by

EUN JAE JUNG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2007

Major Subject: Computer Science

CREATION AND MAINTENANCE OF A COMMUNICATION TREE IN WIRELESS SENSOR NETWORKS

A Dissertation

by

EUN JAE JUNG

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Duncan M. H. Walker
Committee Members,	Jennifer L. Welch
	Rabi N. Mahapatra
	Jiang Hu
Head of Department,	Valerie E. Taylor

December 2007

Major Subject: Computer Science

ABSTRACT

Creation and Maintenance of a Communication Tree in

Wireless Sensor Networks. (December 2007)

Eun Jae Jung, B.S., Myoung-Jee University;

M.S., Oklahoma State University

Chair of Advisory Committee: Dr. Duncan M. H. Walker

A local reconfiguration algorithm (*INP*) for reliable routing in wireless sensor networks that consist of many static (fixed) energy-constrained nodes is introduced in the dissertation. For routing around crash fault nodes, a communication tree structure connecting sensor nodes to the base station (sink or root) is dynamically reconfigured during information dissemination. Unlike other location based routing approaches, *INP* does not take any support from a high costing system that gives position information such as GPS. For reconfigurations, *INP* uses only local relational information in the tree structure among nearby nodes by collaboration between the nodes that does not need global maintenance, so that *INP* is energy efficient and it scales to large sensor networks. The performance of the algorithm is compared to the single path with repair routing scheme (*SWR*) that uses a global metric and the modified GRAdient broadcast scheme (*GRAB-F*) that uses interleaving multiple paths by computation and by simulations. The comparisons demonstrate that using local relative information is mostly enough for reconfigurations, and it consumes less energy and mostly better delivery rates than other algorithms especially in dense environments.

For the control observer to know the network health status, two new diagnosis algorithms (*Repre* and *Local*) that deal with crash faults for wireless sensor networks are also introduced in the dissertation. The control observer knows not only the static faults found by periodic testing but also the dynamic faults found by a path reconfiguration algorithm like *INP* that is invoked from evidence during information dissemination. With based on this information, the control observer properly treats the network without lateness. *Local* algorithm is introduced for providing scalability to reduce communication energy consumption when the network size grows. The performance of these algorithms is computationally compared with other crash faults identification algorithm (*WSNDiag*). The comparisons demonstrate that maintaining the communication tree with local reconfigurations in *Repre* and *Local* needs less energy than making a tree per each diagnosis procedure in *WSNDiag*. They also demonstrate that providing scalability in *Local* needs less energy than other approaches.

DEDICATION

To my parents, wife, and daughter for their patience, encouragement, and love

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my committee chair, Dr. Duncan M. H. Walker. His valuable advice and guidance allowed me to persevere and finish my doctorial degree program. Without his patience and caring for me, I could not have completed my doctoral degree. I am also thankful to my committee members, Dr. Jennifer L. Welch, Dr. Rabi N. Mahapatra, and Dr. Jiang Hu, for their helpful suggestions and careful review of this work. I would like to express my appreciation especially to Dr. Jennifer L. Welch. This research was greatly improved by her helpful guidance.

I thank my parents and parents-in-law for their love and endless support in so many ways. I also thank all my brothers and sisters and other family members. Their prayers encouraged me spiritually.

I deeply give thanks to my wife, Yungah, for her never-ending love, belief, and encouragement. I am also thankful to my sweetheart, Young-Jee, who always brings happiness to me.

I thank God for giving me the strength and belief to finish this program.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

LIST OF TABLES

TABLE Page

I. INTRODUCTION

Wireless sensor networks are seeing increasing usage in sensing applications such as buildings, the natural environment, industry and the military [1][2][3][4]. These networks usually consist of many low-power, low-energy, low-cost sensor nodes with wireless communication links, that are sensing the nearby environment, processing the data obtained from sensing or from other nodes, and communicating necessary data to other nodes or their base station [4][5][6].

Two key communication functions in sensor networks are broadcasting from the base station (sink or root node) to the nodes, and gathering data from some or all nodes to the base station [7]. In energy-constrained wireless sensor networks, message overhead must be minimized since communication consumes most of the energy [8]. This is in contrast to wired networks that are optimized for low latency using the high bandwidth and power available [9].

Each sensor node has a limited radio transmission range, so it must communicate with the base station via intermediate nodes. Even nodes that could reach the base station directly might communicate via intermediate nodes in order to minimize transmission energy [10][11]. A spanning tree has been considered as a communication structure since it requires the fewest messages (energy) to disseminate information from the root to all nodes, and provides a structure for nodes to report their results to the root (base station) [12][13][14][15][16]. To reduce message overhead, a node concatenates the data received from its descendants or aggregates data (e.g., transmits an average value) in a

---

The dissertation follows the style of *IEEE Transactions on Magnetics*.

tree structure [7]. A spanning tree also minimizes message collisions that occur in flooding and other undirected communication schemes [12].

Once a spanning tree is created as a communication structure, it must be dynamically maintained (or reconfigured) by routing around failed nodes and adding new nodes. Routing around failures is required since in the tree structure, each node has only one current parent (path) to the base and some sensor nodes will inevitably fail due to battery depletion or destruction in the harsh environment [3].

The primary goal of this dissertation is creating and maintaining a communication tree to provide a communication structure between wireless sensor nodes and the sink (root) that is energy efficient and reliable against crash node failures. This dissertation describes how communication paths are locally reconfigured by collaboration between nearby nodes, to minimize energy consumption and provide scalability. For that, a new reconfiguration algorithm (*INP*) that uses local relational information in the tree structure is introduced. Unlike other approaches, *INP* does not require periodic message exchanges or continual maintenance of a global metric (e.g. distance from the root) that increase reconfiguration energy consumption. In *INP*, each node knows local tree configuration information (i.e., grandparent, parent, children, and siblings). Based on this information, a node cooperates with nearby nodes to find its new parent when its parent is unavailable. The node tries to find the upper or same level relatives (e.g., great-grandparents, grandparents, uncle/aunt, cousin, or granduncle/grandaunt) among its nearby nodes and connect to one among them, since they already have cycle free paths to reach the root in the tree with having an assumption that *INP* can handle only one

crash fault at a time.

The *INP* algorithm includes joining procedures; so that nodes that are not in the tree (e.g. newly deployed nodes) can join the tree by exchanging messages with neighbors in the tree. Node failures may result in part of the tree being partitioned, so that the partition can no longer communicate with the root. It is undesirable for nodes in the partition to waste energy fruitlessly trying to communicate with the root. By extending *INP* reconfiguration steps to handle partitions, the nodes in the partition can recognize the partition situation and stop sending data until that area is rejoined to the tree like the nodes in *TORA* [17].

The secondary goal of the dissertation is the introduction of system level diagnosis algorithms against crash faults for wireless sensor networks. The impact of node failure on the network capability depends on the number of faulty nodes, the density of nodes, and the specific characteristics of the faulty nodes and the network. If the number of faulty nodes increases without corrective action, the network may ultimately cease to function. To prevent this, a control observer needs to keep track of node status (i.e., diagnosis information). With this information, the network can be reconfigured by bypassing, repairing or replacing faulty nodes when as needed. For example, when an area is partitioned due to node failures, new nodes must be deployed to recover the area [2][16]. Diagnosis information can also help nodes to conserve energy by not sending unnecessary information to faulty nodes.

The methods of obtaining diagnosis information are different depending on the application. In an application where all nodes periodically report data to the control

observer (e.g., a temperature monitoring application [6]), the control observer will learn of node failures from messages that contain information about the dynamic path reconfiguration. When sending data to the control observer, a node will detect a parent node failure through time-outs, and invoke reconfiguration procedures. The faulty node identity is delivered to the control observer by piggybacking on the data. In this type of application, reconfiguration procedures work as a diagnosis algorithm.

In an application where only some nodes are involved in data communication, these dynamic reconfigurations are not enough for the control observer to determine the overall health of the network. In critical or time-sensitive applications, such as tracing moving objects (e.g., tanks or enemies) in a battlefield or a sentry-line defense system on a border, the network must always be monitored and kept healthy by the control observer. Although regular node testing that uses a link level acknowledgement (ACK) of a DATA (a message for testing) in the MAC layer or end-to-end acknowledgement for a testing message of the application layer consumes more energy, each node can periodically check neighboring node(s) and communicate any failures to the control observer.

Based on diagnosis information obtained from regular testing and dynamic reconfiguration procedures, an appropriate corrective action can be taken to maintain the communication structure of the network. Two new diagnosis algorithms (*Repre* and *Local*) that use *INP* reconfiguration steps are introduced in this dissertation. In *Repre*, all nodes report diagnosis information to the root. When the network size grows, *Repre* is extended to *Local* to provide scalability by reducing energy consumption and

communication overhead.

This dissertation is organized as follows. Section II describes previous work. Section III describes *INP*, the reliable and dynamic reconfiguration procedures to maintain the communication tree. In section IV, *INP* is analytically evaluated and compared with the single path with repair routing (*SWR* [18]). In section V, *INP* is evaluated with *SWR* and *GRAB-F*, the fixed transmission power version of *GRAB* [19] through ns-2 simulations. Section VI introduces the *Repre* and *Local* diagnosis algorithms and analytically evaluates them along with the *WSNDiag* crash fault identification algorithm [4]. Conclusions and future work are described in section VII.

II. REVIEW OF PREVIOUS WORK

A. Routing algorithms for wireless sensor networks

Many different routing approaches have been introduced in different environments and there is not a single approach that we can say always gives better performance than any other approaches. A good routing strategy for a certain network environment comes from considering the special characteristics of the environment. For example, algorithms that consider moving nodes [15][20] require frequent message exchanges to maintain the communication structure, and so use unnecessary energy when nodes are fixed or slowly moving, as in a sensor network.

In this section, some of the existing routing approaches for wireless sensor networks and their shortcomings are reviewed and suggested directions for wireless sensor networks that can save communication energy are introduced. Routing approaches are classified by whether or not paths are dynamically reconfigured. This permits direct comparison with the proposed approach.

1) Approaches without dynamic path reconfiguration

There are single path [21][22] routing algorithms without path reconfiguration. In the rumor routing algorithm [21], a damaged route can be recovered by an agent of another event, if available. But this situation is not always guaranteed for the broken paths. In [21], with increasing number of events, the cost of maintaining routing information in each node rises [23]. In the directed diffusion algorithm [22], periodic flooding is needed for maintaining the path. These algorithms can be used in a benign environment to

minimize communication energy. But in a harsh environment, message delivery can easily fail due to node failures [18].

To achieve a higher message delivery rate, routing algorithms that maintain multiple paths without dynamic path reconfigurations were introduced [22][24][25]. Directed diffusion algorithm [22] can have either a single path or multiple paths, depending on the path number(s) being reinforced from the sink [22]. Energy aware routing algorithm [24] always uses a single path to send information among multiple paths maintained for path energy balancing. To maintain multiple paths, infrequent localized flooding from destination to source is performed. In [25], disjoint and partially disjoint braided multi-path routing schemes were introduced, that use pre-routed alternate path(s) when the primary path is broken. With these approaches, the energy consumption due to periodic flooding in [22][24] can be reduced. But when increasing node failures break all multiple paths, including the alternate path(s), flooding must be used to reestablish the paths [25].

The same information can be redundantly delivered through multiple paths [22][24][25]. Determining the ideal number of paths to balance delivery ratio, energy consumption, and network congestion is difficult [18]. Using the wrong number of paths can cause unnecessary energy consumption or low delivery ratio [18].

Unlike the multiple path approaches [22][24][25], GRAdient Broadcast (*GRAB*) [19] uses interleaved paths that are not fixed in advance, but are created dynamically and form a mesh structure when a message is delivered to the sink node. In *GRAB*, each node that receives a message decides for itself whether to send the message, how many

neighbors to send it to (by adjusting its transmission power), or whether to drop the message. Before data transmission, each node already knows its minimum cost to the sink through advertisement (*ADV*) packets initiated by the sink. Only the packet-receiving nodes that have smaller minimum cost than the forwarding node's minimum cost forward the packet. Packets in all other receiving nodes are dropped. But in densely deployed sensor fields that have many low-cost paths, this packet dropping approach is not sufficient to prevent unnecessary message redundancy that causes unnecessary energy consumption and message loss due to collisions. To limit the number ("width") of the forwarding paths further, *GRAB* used a credit (α) that is assigned to the message at the source, together with the minimum cost (C_{source}) of the source. The maximum value that can be used for delivering each message is $\alpha + C_{source}$. Among those that have smaller costs than the cost of the forwarding node, only the nodes that have sufficient credit can forward the message [19]. A message flow example of *GRAB* is shown in Fig. 1. The white nodes indicated by arrows are the packet-receiving nodes that have larger minimum cost than the forwarding node's minimum cost.

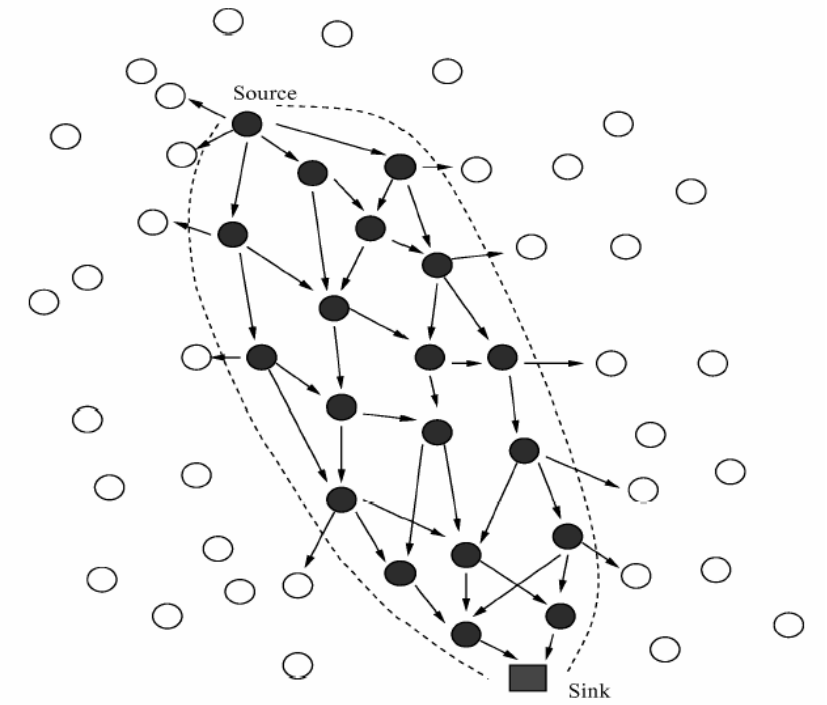


Fig. 1. An example of GRAB flow from a source to the sink [19].

2) Approaches with dynamic path reconfiguration

For ad hoc wireless networks, there are some single-path routing with path repair algorithms [26][27]. These approaches detect a hop failure (due to a failed link or node) and then report it to the sender, which must find an alternative path to the destination. This is undesirable when there are many hops between sender and destination [18]. It is particularly undesirable when the root is broadcasting to all nodes, or all nodes are gathering information for the root.

There has been limited research on routing path reconfiguration in wireless sensor networks to avoid node failures. The “big base station” centralized approach [16] maintains a global view of the network in a powerful base station for wireless sensor

networks. The base station builds and maintains the network routing topology by using the neighboring information received from each node, and sets paths by communicating directly to the nodes. Node failures require the base station to iteratively localize failed nodes and determine alternate paths. Much iteration can be required before all reachable nodes are reconnected to the network. This approach takes advantage of the computing and broadcast power of the base station to control all maintenance procedures, but does not scale.

Local reconfiguration approaches are more appropriate for scalability and minimum communication energy in wireless sensor networks, since sensors generally collaborate with their neighbors to produce valuable and reliable data [11].

The single path with repair (*SWR*) scheme [18] uses local path repair, in which nearby nodes are used to find an alternate path to the destination, backtracking as necessary, while preventing loops. On the path used for information delivery, the node (called a *pivot* node in [18]) that has a faulty next immediate node (called *downstream* or *parent* node), initiates the path repair procedure by broadcasting a *Help Request (HREQ)* control packet [18]. The pivot node determines the best alternative node of the faulty node among the neighbors that reply with a *Help Response (HREP)* control message in response to *HREQ*. The best alternative is the node that has the lowest cost *C* among them. When the number of hops is used as the cost metric, a neighbor that has the fewest hops to the root is chosen.

A *HREP* message is either initiated from a neighbor of the pivot node directly or it is relayed to the pivot node through a neighbor after initiating at a downstream node of the

neighbor indirectly. The latter happens when a *HREQ* message is relayed to the downstream node of a neighbor, since the neighbor has higher cost than the pivot node. When the downstream node that has equal or lower cost than the faulty node receives the *HREQ* message, it replies with a *HREP* back along the path to the pivot node. Relaying the *HREQ* message to downstream nodes is limited (e.g., 3) to prevent a loop. When the pivot node cannot find any alternative node of the faulty node, it returns information back to the source node, with each node along the reverse path attempting to find an alternative path [18].

Fig. 2 shows a reconfiguration situation in *SWR* [18]. When node *F* becomes faulty, node *A* broadcasts an *HREQ* message. Nodes *B*, *G*, and *H* discard it since node *B* has same faulty node *F* as its parent and nodes *G* and *H* have the pivot node *A* as their parent. When node *E* receives the *HREQ* message, it relays it to its parent node *C*, since its cost is higher than that of node *A*. Node *C* again forwards this message to node *D* since its cost is not lower than the cost of node *A*. Since nodes *D* and *F* have the same cost *k-1*, node *D* replies with an *HREP* message to node *A* through the reverse path direction (i.e., $D \rightarrow C \rightarrow E \rightarrow A$). When node *D* receives an *HREQ* message directly from node *A*, it replies with an *HREP* message to node *A*. When node *C* receives an *HREQ* message directly from node *A*, it forwards the *HREP* message issued from node *D* to node *A*. Node *A* chooses node *D* as its downstream node when it receives *HREP* messages from nodes *E, C,* and *D* since node *D* has the lowest cost among the nodes.

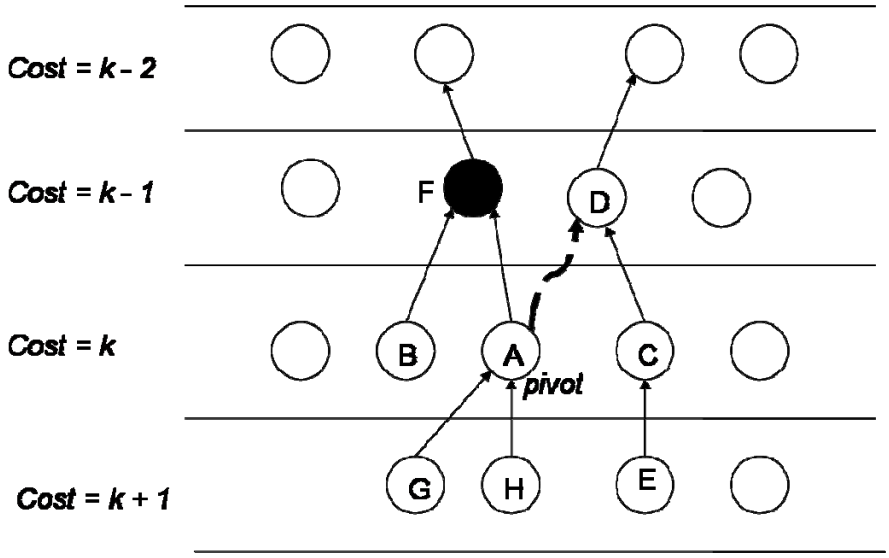


Fig. 2. An alternate path selection in *SWR* approach.

Since the *SWR* algorithm uses a global metric, nodes upstream from the pivot node must update their cost *C* after the local path repair procedure. This requires communication energy. The cost update can be postponed to minimize this communication [18], but outdated cost information could cause a repair to create a loop, requiring further repair or causing message delivery failure.

3) Goal approach

We propose a new single path with reconfiguration approach that only needs local relative information (e.g. parent, grandparent) in each node. Since wireless sensor networks consist of stationary nodes, this information is infrequently updated, in contrast to the nodes in MANETs. A constant number of message exchanges among fault-free neighbors is enough for many cases of path reconfiguration. Unlike the *SWR* approach

[18] that uses a global metric, the neighborhood information does not need to be updated incrementally among all nodes.

Since the routing structure of the proposed approach is a tree, the reconfiguration of the routing paths is related to tree maintenance. Thus we also review prior work in tree creation and maintenance.

B. Spanning tree creation and maintenance

The communication structure using a spanning tree or directed acyclic graph (*DAG*) has been used for many different applications in many different environments. Through this acyclic graph structure among nodes, data can be multicast, and routes can be found [28]. In this subsection, we roughly categorize and review the prior tree maintenance approaches. We also evaluate the shortcomings of these approaches when applied to wireless sensor network environments and suggest an approach for these environments.

Before introducing the prior tree maintenance approaches, we introduce an example that explains the merits of a tree communication structure by comparing it with a simple flooding approach in wireless sensor network environments. Fig. 3 shows a 5-node wireless sensor network with the transmission range around each node, and the communication graph for the network. Based on Fig. 3, Fig. 4(a) shows when node 4 sends a message to the sink node 1 in a tree structure through a path ($4 \rightarrow 2 \rightarrow 1$) that connects it to the sink. The flow of messages in a simple flooding based dissemination approach is shown in Fig. 4(b).

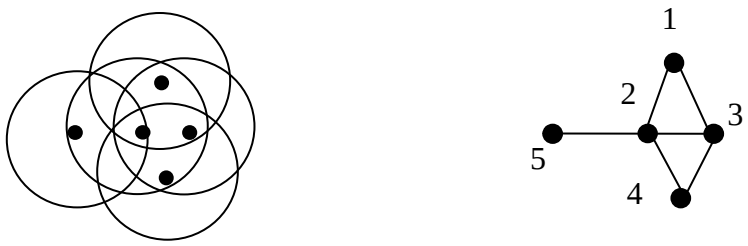


Fig. 3. 5-node wireless sensor network and its communication graph.

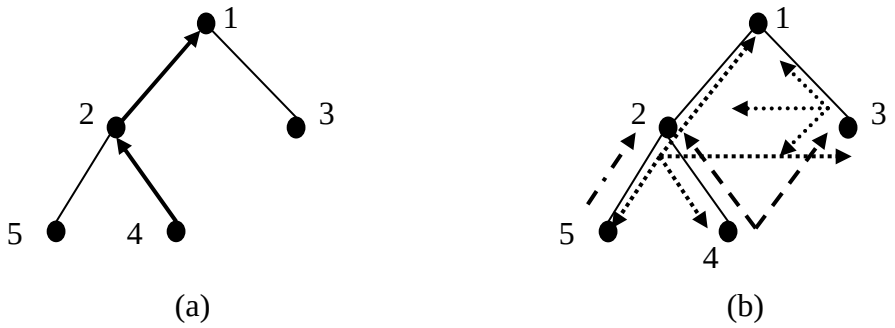


Fig. 4. Information dissemination with tree and flooding.

There are redundant messages in simple flooding because each node sends a message to all of its neighbors without checking whether a neighbor has already received the message from other nodes (i.e., implosion problem) [12]. When each node has many neighbors, and the transmitter ranges have a large overlap (i.e., overlapping problem) [12], many redundant messages result.

1) Spanning tree construction per each use

In centralized multi-hop lightweight time synchronization (*LTS*) [29], a spanning tree is made whenever time synchronization is needed for wireless sensor networks [29]. In a

distributed approach that maintains the fault status of all nodes within each node (*WSNDiag*) [4], a tree is made whenever an observer initiates this algorithm for identifying crash faulty nodes in a wireless sensor network. (A crash fault is one in which the node simply stops working and goes silent). This algorithm is simple, but would not be a good approach for frequent use in stationary sensor networks since it uses much redundant energy to rebuild the whole tree for each use.

2) Maintaining total knowledge of network

The spanning tree can be built once and then maintained. Various tree reconfiguration approaches have been introduced for different environments. Some of this work considers the problem of new nodes joining the network and the network being partitioned and then rejoined. Many tree maintaining algorithms have been developed that use a global view of the network for maintaining the spanning tree. Many of them attempt to maintain a minimum spanning tree (*MST*).

a) Maintaining a minimum spanning tree (MST)

There were classical algorithms that form and maintain a minimum spanning tree (MST) using total knowledge of a network. An algorithm that updates MST and shortest paths when graph parameters are changed was given in [30]. When a new node is added to the MST of an *n*-node graph, this algorithm updates the MST with *O(n)* comparisons and *O(n)* storage [30]. An algorithm for maintaining a minimum spanning forest in a dynamic plane graph was introduced in [31]. The forest is maintained under edge weights, and insertion and deletion of vertices and edges. A dual edge-ordered dynamic

graph [31] was used with a primal graph to maintain the minimum spanning forest. Creating and maintaining a dual graph [32], corresponding to a given plane graph, is difficult without location information for all nodes.

b) Centralized approaches

The “big base station” centralized approach [16] maintains a global view of the network in a base station for wireless sensor networks. In the algorithm, the big base station does not attempt to maintain a MST but gives alternate paths around failed nodes to the nodes that need them. In wireless sensor networks that consist of many nodes, scalability cannot be provided with this approach.

c) Distributed approaches

Some distributed algorithms maintain the global view of the network in each node. In [33], a spanning tree is used for improving database maintenance in dynamic networks where edges may fail or recover. Each node continuously updates a dynamic data structure that has the tree replicas of all the nodes in the network by communicating with other nodes about any changes. Each node is assumed to know the content of the local memory of all of its neighbors, and for each error sends a message to the neighbor that has the error [33]. When a tree disconnected by an edge failure is merged into another tree, it locates its minimum-weight outgoing edge to other trees through the dynamic data structure. When two trees have the same minimum-weight outgoing edge to each other, the trees are merged through it.

It is impractical to maintain total knowledge of the network in a special node or in

each node, since it requires much communication and does not scale with large networks.

In [28], a node failure partitions the acyclic graph into one or more subgraphs (or fragments). These fragments are coordinated to recreate a complete acyclic graph. Physically sufficient network connectivity is assumed in the presence of node failures so that each fault-free node can reach every other node. Whenever a node detects the failure of a neighbor, it starts a reconfiguration by flooding a *Reconfig(node_list, frag_id)* message having *frag_id* and *node_list* with its own ID. The nodes on the path that the *Reconfig* message follows are added to *node_list*. Links that are needed for combining fragments are accepted as edges in the combined acyclic graph by resolving contention among fragments, avoiding cycles and message loops. When two partitioned fragments are joined, the pre-established higher-ranked *frag_id* or randomly selected *frag_id* among them can be used. This algorithm does not consider individual nodes joining (reseeding) the network. Since flooding is used and every node participates in reconfiguration, the energy consumption of this algorithm makes it unsuitable for wireless sensor networks.

3) Maintaining a global metric

There were many other distributed approaches that do not need a total view of the entire network in each node. Instead, global information (e.g. distance to the root of the tree) is locally maintained in each node and used for tree maintenance. By collaborating with its neighbors, a node's information is incrementally updated among other related nodes. In this dissertation, these are called incremental approaches.

a) Various incremental approaches

In self-stabilizing spanning tree construction algorithms [34], each node updates its new parent with the neighbor that has minimum distance value obtained by regularly exchanging information with neighbors. In [35], an arbitrary spanning tree is created and maintained for a dynamic network where edges may fail or recover. Each node keeps three variables: its parent, distance from the root, and its current root. Those are maintained when edge removal or addition occurs, by exchanging messages (e.g. *M-message (root, distance*) [35]) among neighbors and used for reconfiguring the spanning forest. In [36], nodes in a computer network are maintained in a rooted spanning tree (RST) as long as the nodes remain connected in the presence of a finite number of fail-stop failures and recoveries. Each node maintains three values: parent, root, and color. In this fully distributed, nonmasking fault-tolerant protocol, the spanning forest is merged into a spanning tree rooted at the node that has the highest index [36]. This protocol can tolerate a finite number of faults during tree reorganization caused by previous faults, but at the cost of high message overhead. The time complexity of RST is *O(N)* rounds where *N* is the number of fault-free nodes. The single path with repair (*SWR*) routing scheme [18] for wireless sensor networks that was introduced in the previous subsection also belongs to this approach.

b) Link reversal approaches

There were also link reversal algorithms among the incremental approaches. In *TORA* [17], a loop-free routing algorithm for mobile ad hoc networks (MANETs) was introduced. It adds a partition detecting procedure to *GB* [37] that generates loop-free

routes in the networks with frequently changing topologies due to link failures. Full and partial reversal methods were introduced in *GB*. Fig. 5 [37] shows an example of simple full reversal method. Initially all flows are directed to the destination *D* without a cycle. With a link *A* to *D* failure, node *A* has incoming paths without an outgoing path, denoted by *R* in Fig. 5. Then all incoming paths of node *A* are reversed. In each iteration, the node that does not have an outgoing path reverses all its incoming paths and finally an acyclic graph is made.

Like *GB*, all nodes in *TORA* are totally ordered with different values by having incremental updates among nodes and thus each node has a its height within the tree. The direction of links is always from higher to lower and thus cycles can be prevented. Whenever a node (except the destination node) does not have at least one outgoing link, it reverses its link directions to form an outgoing link. A destination-oriented *DAG* is finally produced by having a destination that has the lowest height. Unlike *GB* [37], when the network is partitioned, all invalid routes among the nodes in the partition are erased to stop sending unnecessary messages. This conserves energy in the partition. Based on *TORA*, an approach that has a leader per partition was introduced for MANETs [38]. Whenever a node detects partition, it elects itself as the leader of the nodes in the partition and announces it to its neighbors. This information is disseminated until all nodes in the partition have the new leader. When two partitions are combined together due to a new link joining them, the leader that has the smallest ID becomes the leader of all nodes in the combined graph [38]. In [38], algorithms were introduced for both a single link (topology) change and a change that occurs before recovering from the

previous change. Like *TORA* and *GB*, cycles are prevented since paths are made only from higher to lower heights [17][37][38].

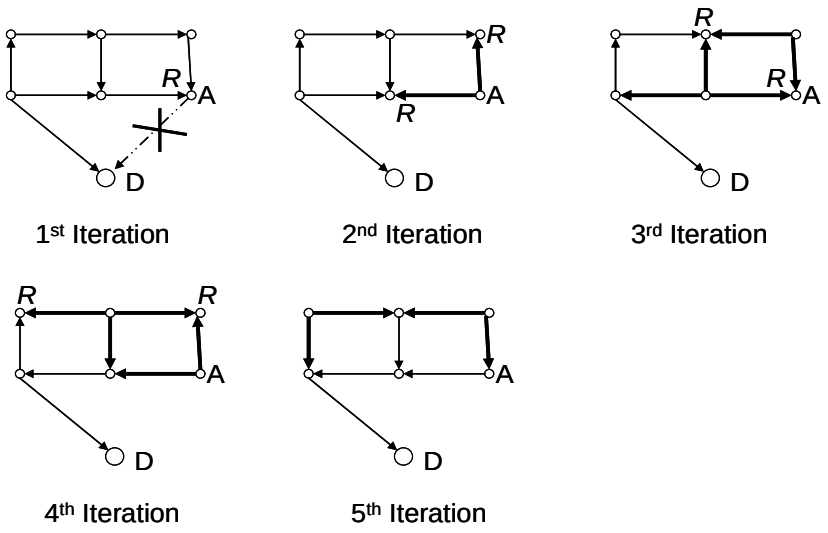


Fig. 5. Full reversal method in *GB* [37].

As shown in the previous examples, most incremental approaches were used for dynamic networks with frequent network topologies changes, such as MANETs. In those environments, frequent message exchanges among neighbors are required to update global information. Outdated global information could cause a repair to create a loop, requiring further repair or causing message delivery failure. But in stationary wireless sensor networks with more severe energy constraints, these approaches can cause high energy consumption.

The features of the above prior work can be roughly categorized as: constructing a

new spanning tree per use without maintaining it; updating MSTs with total knowledge of a network; updating a global view of the network in a central node (centralized) or in each node (distributed); or updating global information locally and incrementally in each node. Most of these approaches are not suitable for direct application to static wireless sensor networks, due to high message overhead that causes high energy consumption.

4) Goal approach

Maintaining a single routing path in a tree structure by using only local relative information (i.e. parent, grandparent, etc.) in each node was already suggested in the previous subsection. The reconfigured tree will be a spanning tree that is not necessarily minimum, in order to avoid the high cost required to achieve an MST. Furthermore, only the local information is used when considering the problems of new nodes joining the network and the network being partitioned and then rejoined.

III. RELIABLE AND DYNAMIC RECONFIGURATIONS OF THE COMMUNICATION TREE

In this section, a new energy efficient and reliable single path routing algorithm is described that uses only local relative information for dynamic reconfiguration of broken paths caused by crash faults in wireless sensor networks. We focus on failure during communication from the sensor nodes to the sink or root. The following assumptions are made for the approach:

- The sensor network consists of randomly distributed stationary nodes that have unique node identifiers with omni-directional antennas, with a fixed communication range.
- All links are bidirectional, that is, if a node can receive messages from another node, the other node also can receive messages from the node. If one node has a longer transmission range than a neighbor, a link will not exist between them, since messages cannot be acknowledged in both directions.
- The root node of the tree (also known as base station or sink node) *R* knows its identity. Thus, there is no leader election for the root. The root will typically have a different implementation than the sensor nodes, since it must communicate with the control observer that manages the network and provides statistical information for the users in the outside world.
- Enough nodes are deployed in the field so that most of them will be able to join the communication tree, that is, they will initially have several neighbors within their transmission range.

- Only crash faults are considered. A node fails by going silent. It cannot have malicious or intermittent behavior. If a node is found faulty by a neighboring node, it is also found faulty by all other neighbors. A crashed node can later recover (e.g. battery recharges). This is treated the same as seeding a new node into the sensor network.
- A single crash fault occurs. Another crash fault cannot occur until reconfiguration from the first one is completed. Relaxation of this assumption will be discussed in a following subsection.
- Fault detection is initiated by a child in the tree, when attempting to send a message to the root. It can be done by link level acknowledgement in the MAC layer, as in IEEE 802.11 [39] or via a validation transaction, such as end-to-end message acknowledgement and timeout in the application layer. The exact fault detection mechanism is outside the scope of this dissertation.
- Two kinds of communication methods are used. One is cheap but unreliable broadcast and the other is expensive but more reliable unicast. A MAC layer such as IEEE 802.11 is available that provides reliable message unicast capability.

A. Spanning tree creation

After sensor nodes have been deployed, an initial spanning tree *T* must be formed, having the *sink* node as the root of the tree. The root broadcasts a *PARENT* control packet with its ID and that of its parent (*NULL* in the case of the root). Receiving nodes

that do not have a parent node set their parent and grandparent information and broadcast their own *PARENT* control packet. Other receiving nodes that already have a parent set their children, sibling, and neighbor information based on the *PARENT* control packets. The process stops when all reachable nodes have a parent set. Even though broadcasts are unreliable, all reachable nodes will eventually join the tree. If they miss a *PARENT* broadcast, they will time out and broadcast an *I Need Parent (INP)* control message, to which a neighboring node can respond. This is described in the following subsection. The *INP* mechanism is also used for new nodes joining the network.

In the approach described here, each node maintains parent (*p*), grandparent (*gp*), children (*chd*) and sibling (*sibs*) information. In order to increase the success of reconfiguration, this can be extended to *K* ancestors, but at the cost of additional communication to maintain the information.

Fig. 6 shows how the paths of the 5-node wireless sensor network that is shown in Fig. 3 are made with message flows. Fig. 7 shows the resulting tree. Node 1 is the root of the tree.

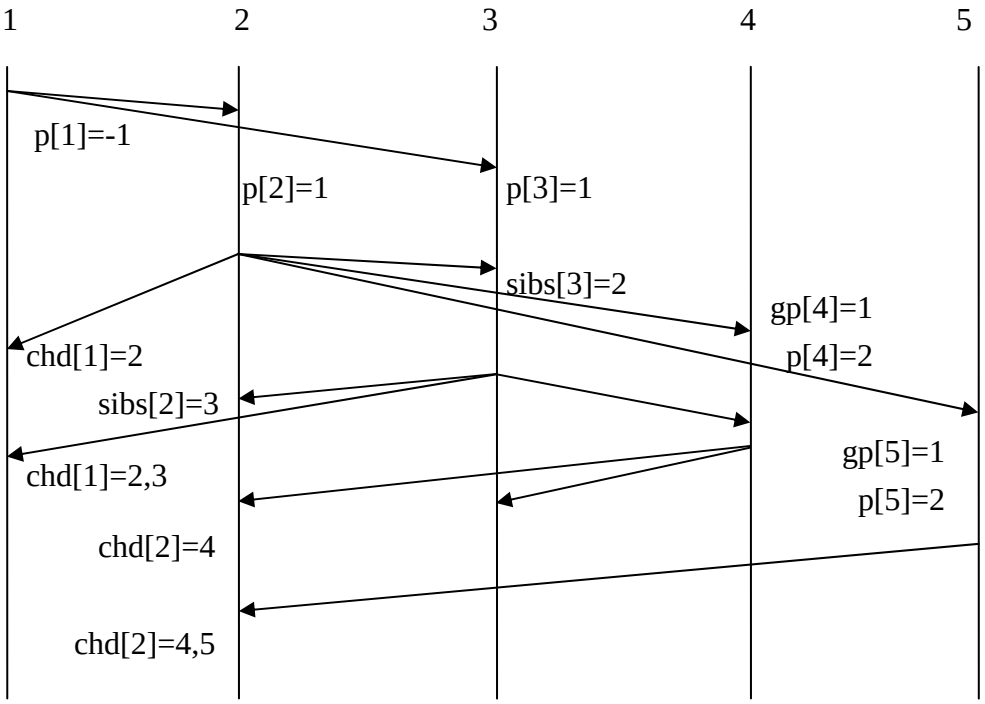


Fig. 6. Message sequence for tree creation.

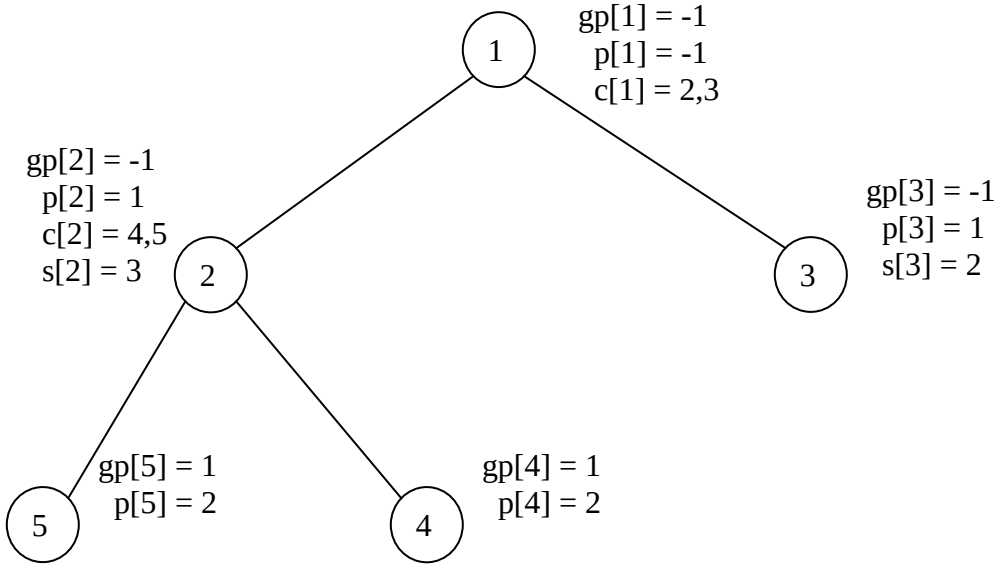


Fig. 7. Final tree made.

B. Spanning tree maintenance

The tree *T = (V, E)* consists of *n* vertices (nodes) and *n-1* edges that connect those vertices. When a faulty node is found by a child on the path during message propagation from a source to a destination, the path must be reconfigured. In case of a failure, the detecting node finds a new parent node and sends the message via the new parent. For this reconfiguration, each node will have only local network information; *p*, *gp*, *chd*, and *sibs* that is obtained when the spanning tree is created and updated during maintenance.

For reconfiguration, some control packets will be used locally. There are three flow directions: *HIGH, LOW,* and *UNKNOWN*. When a node finds a neighbor that guarantees a cycle free path to the root, it sets the neighbor relationship to *LOW*. The root node is the lowest one in the network. When a node finds a neighbor that is either a descendant of the node that causes a cycle or a descendant of its sibling node, it sets the neighbor relationship to *HIGH*. If a neighbor is not known as *HIGH* or *LOW*, it is set to *UNKNOWN*. When a data message arrives at a node that needs a new parent, our reconfiguration algorithm (called the *INP* algorithm in the remainder of the dissertation) is initiated after queuing the message in the node. This algorithm has several sequential steps (*INP*, *CNFCF*, *ICNYP*, *INI*, and *PFIND*) as shown in Fig. 8. Each step is run with a time limit, with the algorithm advancing to the next step if reconfiguration is not completed within the given time. When a new data message arrives while a reconfiguration step is running, the message is queued and sent to the new parent that is obtained after completion of the reconfiguration. This approach is different from *SWR* [18], where a new reconfiguration is initiated whenever a message arrives at the node

that needs a new parent. *SWR* uses a message based reconfiguration mechanism while *INP* uses a node based reconfiguration mechanism. These different mechanisms will be compared in section V. The following subsections describe each step in the *INP* algorithm.

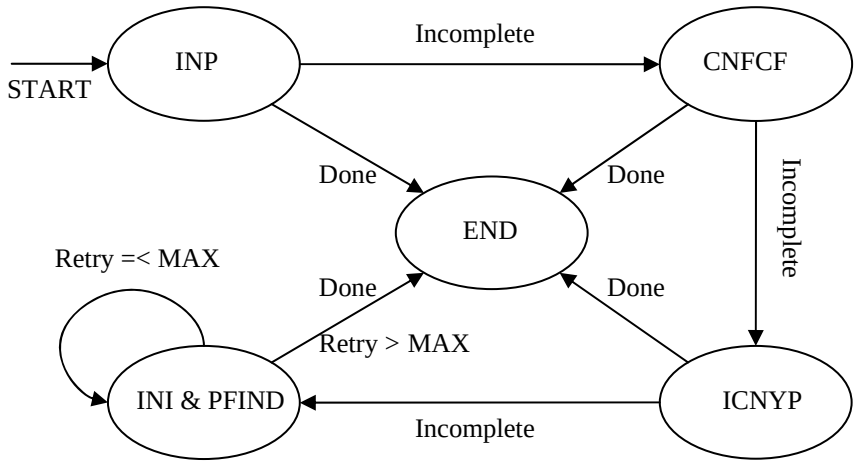


Fig. 8. Overall reconfiguration steps in *INP*.

1) INP

Whenever node n_i starts a reconfiguration for a new parent due to the failure of its current parent n_{i+1}, it broadcasts an *INP(init, myID, init.p, init.gp)* control packet. *INP* stands for *I Need Parent*, *init* is the ID of the initiator of the reconfiguration, and *myID* is the ID of the sender of the *INP* control packet. Initially, *init* and *myID* are node n_i. When any node n_j that is not a child or a sibling of node n_i receives the *INP*, it checks if it can provide a *LOW* direction for node n_i based on its local information and the information in the *INP*. If so, it unicasts an *ACKINP(init, myID, myID.p, caseNUM)* control message to node n_i. *ACKINP* stands for *Acknowledgement of I Need Parent*.

As an example, Fig. 9 shows a situation when node *A* finds its new parent and Fig. 10 shows the message sequence. Since node *S* has parent *D* that is a grandparent of node *A*, it provides a cycle free path.

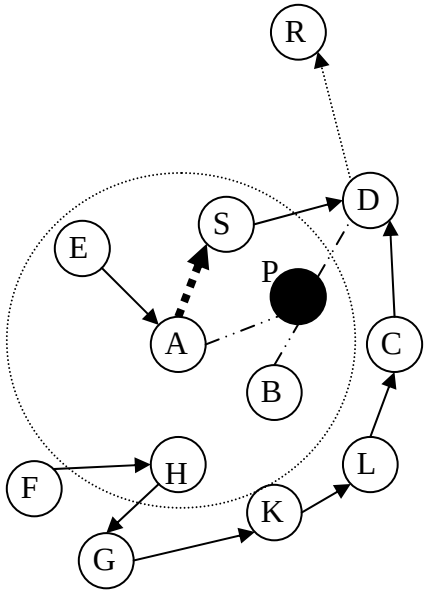


Fig. 9. When node *A* can find its new parent using *INP-ACKINP*.

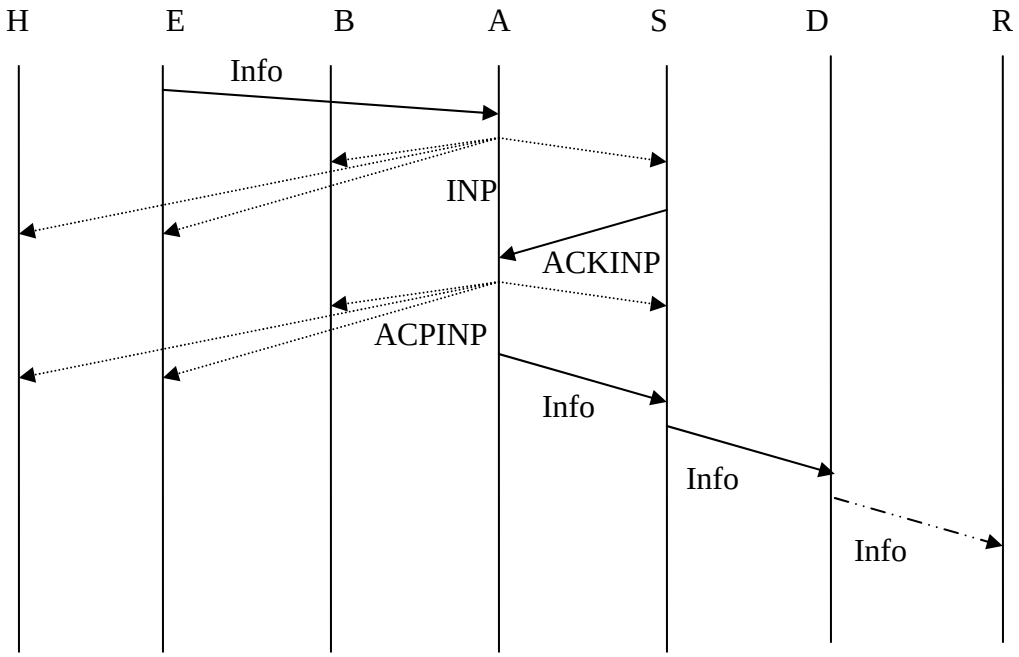


Fig. 10. Message sequence for Fig. 9.

Fig. 11 shows five cycle free reconfiguration cases and each case is shown in Figs. 12 to 16. Fig. 9 was an example of case 3 (Fig. 14).

```
receiveINP(init, i, init.p, init.gp)
{
   if (init.gp ∈ myID.chd) {                                          /* Case 1 */
          send ACKINP(init, myID, myID.p, 1);
   } else if (init.gp == myID) {                                      /* Case 2 */
       send ACKINP(init, myID, myID.p, 2);
   } else if (init.gp == myID.p) {                                    /* Case 3 */
       send ACKINP(init, myID, myID.p, 3);
   } else if ( (init.gp == myID.gp) && (init.p != myID.p) ) {         /* Case 4 */
       send ACKINP(init, myID, myID.p, 4);
   } else if (init.gp ∈ myID.sibs) {                                  /* Case 5 */
        send ACKINP(init, myID, myID.p, 5);
}
```

Fig. 11. Cycle free reconfiguration cases.

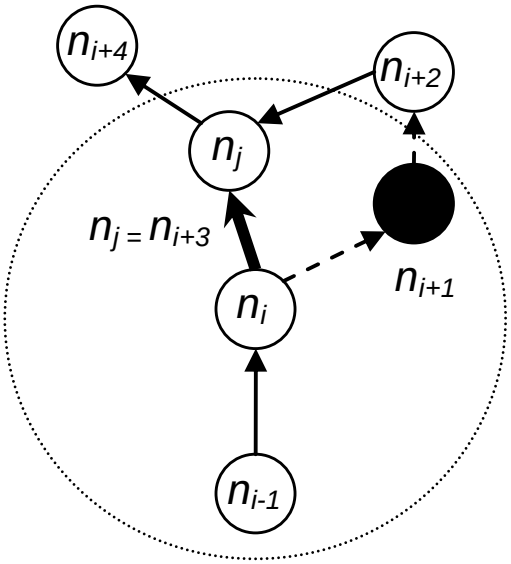


Fig. 12. Case 1 – When node n_i has a neighbor n_j that has the grandparent of node n_i (i.e. n_i+2) as its child. That is, n_j is a great-grandparent of node n_i.

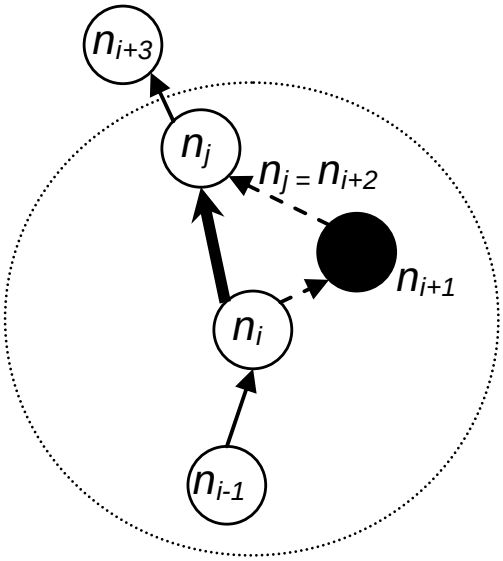


Fig. 13. Case 2 – when node n_i has a neighbor n_j that is its grandparent $n_{i+2.}$

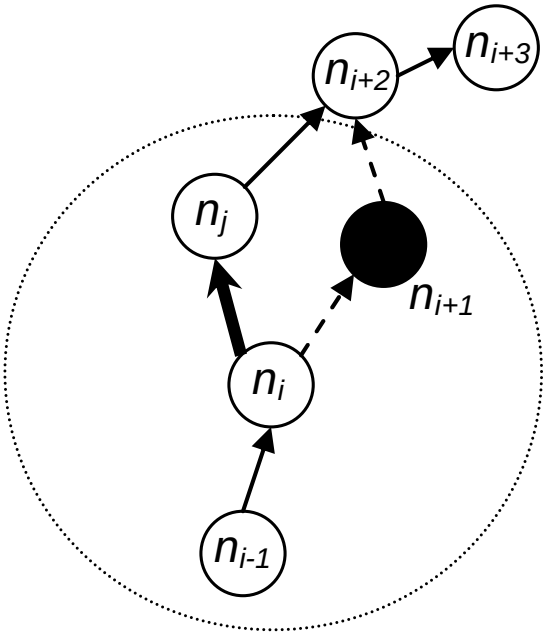


Fig. 14. Case 3 – when node n_i has a neighbor n_j that has the grandparent of node n_i (i.e. n_{i+2}) as its parent.

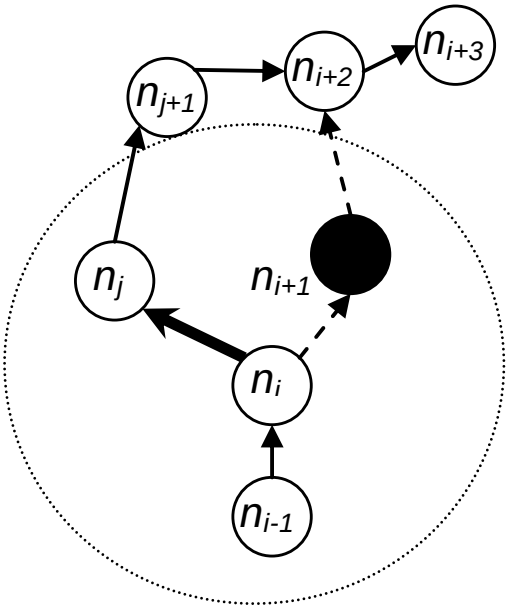


Fig. 15. Case 4 – when node n_i has a neighbor n_j that has n_{i+2} as its grandparent.

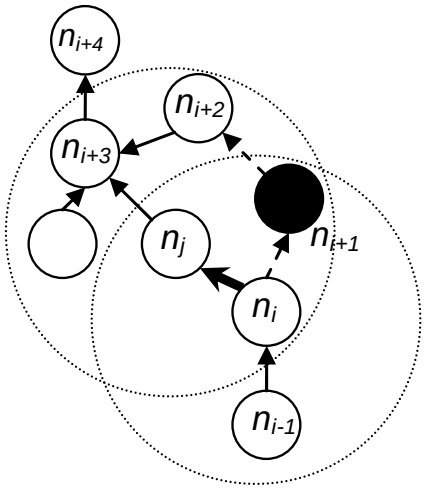


Fig. 16. Case 5 – when node n_i has a neighbor n_j that is sibling of node $n_{i+2.}$

Several nodes may send an *ACKINP*. Then node n_i selects one of the nodes n_j as its new parent by broadcasting an *ACPINP(myID, myID.p, myID.gp, caseNUM)*. *ACPINP* stands for *Acceptance of I Need Parent*. Node n_j adds node n_i to its children list. Then the data message that was held at node n_i is delivered to node n_j.

When node n_i broadcasts the *INP*, some siblings of node n_i may receive this message and thus learn their parent is faulty. At this step, the sibling will wait until node n_i finds a new parent and then set node n_i as its parent after receiving *ACPINP*. In this way, some siblings can find a new parent without their own search, saving energy. But in a message based reconfiguration mechanism such as *SWR* [18], each node that has a faulty parent performs its own search for a new parent when it receives a data message, increasing energy consumption. Fig. 17 shows the next step of Fig. 9. In Fig. 17, node *B*, a sibling of node *A*, sets node *A* as its parent after receiving *ACPINP* from node *A*.

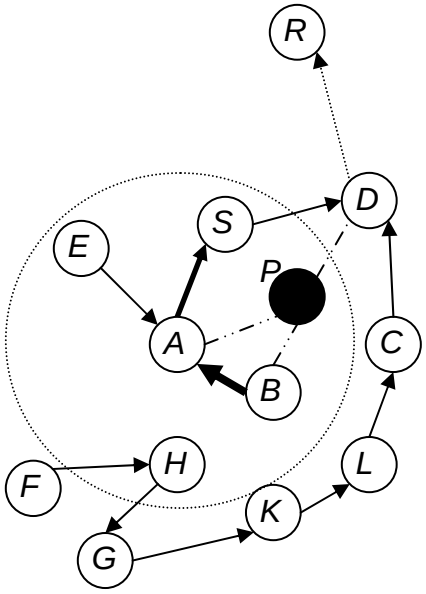


Fig. 17. Sibling node *B* of node *A* can find its new parent.

The *ACPINP* also helps the children of node n_i to find their new grandparent. In Fig. 17, when node *E* receives *ACPINP* from node *A*, it learns of its new grandparent, node *S*. Optionally, if *ACPINP* provides *myID.gp* information (i.e., node *D*), node *E* can also learn its great-grandparent (*ggp*) from the message (*myID.gp* = *D*). When this extended ancestor information is used, the success of reconfiguration can be increased.

In addition to cases 1 to 5 above, a case 6 was introduced in [40], shown in Fig. 18. This is the case when node n_i has a neighbor n_j that has a grandparent (n_{i+2}) of node n_i as its neighbor.

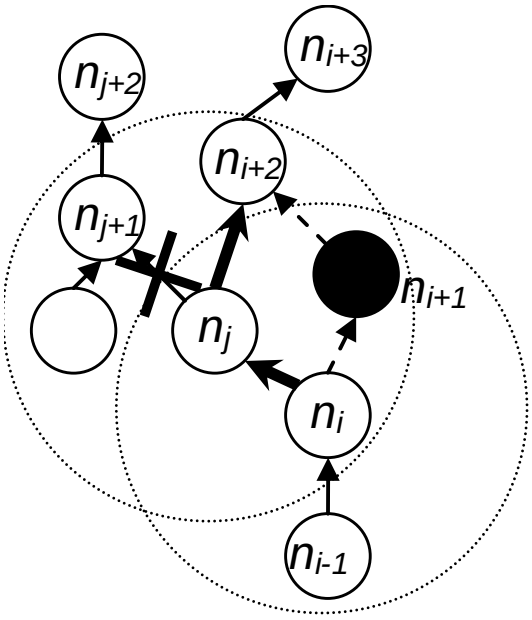


Fig. 18. Case 6 – when node n_i has a neighbor n_j that has n_{i+2} as its grandparent.

Case 6 does not always provide a cycle free path. Unlike the above five cases, node n_j changes its own parent to node n_{i+2}. This can cause a cycle since node n_{i+2} can be a descendant of node n_j. The same situation can happen when node n_j keeps its own parent. The parent of node n_j can be a descendant of node n_i.

Figs. 19 and 20 show two actual situations that caused cycles during simulation. Fig. 19 shows a cycle that happened when node n_j changed its own parent to node n_{i+2} as in case 6. Fig. 20 shows a cycle that happened when node n_j kept its own parent.

Fig. 19 shows the situation when node 52 became faulty. When node 42 chose node 62 as its new parent after node 62 sent *ACKINP* to it, since node 42's grandparent node 61 is its neighbor and when node 62 changed its current parent with 61, a cycle path was created. In this case, node 62 avoids a cycle by not changing its parent.

Fig. 20 shows the situation when node 46 became faulty. When node 28 chose node 36 as its new parent after node 36 sent *ACKINP* to it, since node 28's grandparent node 56 is its neighbor, a cycle path was created. In this case, node 36 avoids a cycle by changing its parent with node 28's grandparent node 56 as its new parent.

Although case 6 can be used with a cycle detection and removal method, the extra complexity of the algorithm is not worthwhile, since the later steps in the algorithm can handle the cases not handled by cases 1 to 5.

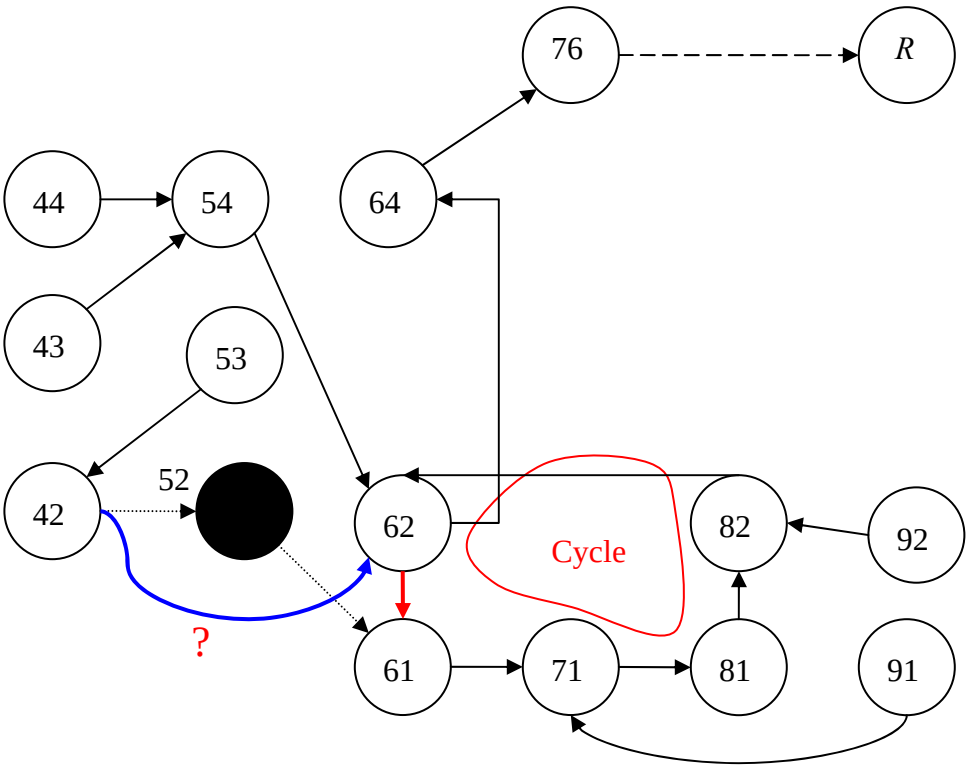


Fig. 19. A cycle occurs when node n_j changes its parent to node n_{i+2} in case 6.

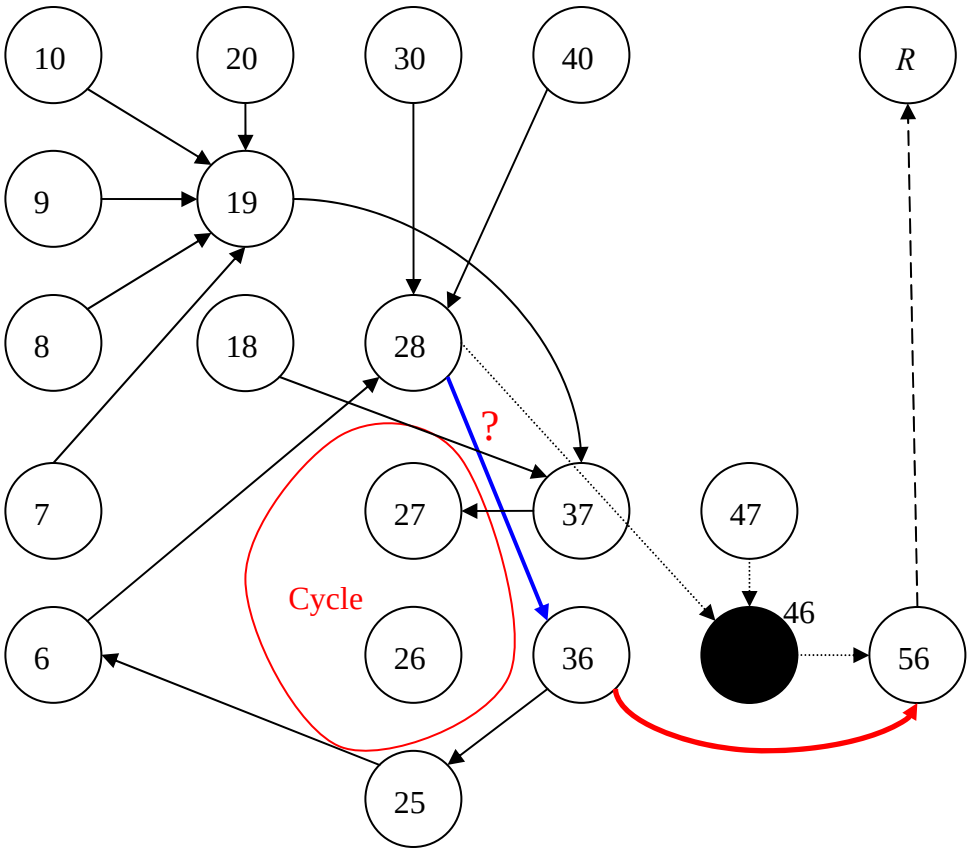


Fig. 20. A cycle occurs if node n_j keeps its own parent in case 6.

Lemma 1. Given a faulty node in a spanning tree, the new path to the root node *R* made by local reconfiguration using a case from 1 to 5, is loop free.

Proof. We know that water on the mountain flows down into a valley and a ball on the inclined plane rolls down to the bottom. We also know that these natural flows are loop free. Each path in a spanning tree is like a water flow on the mountain. Information from nodes flows down into the root, *R*. Without loss of generality, we can say that a new path made by a reconfiguration case is loop free if the initiating node of the reconfiguration is directly or indirectly connected to a lower or same height node on the current path when

the height of *R* is 0.

Both case 1 and 2 provide direct connections to a lower height node on the path. In case 1, the initiator is directly connected to its previous great-grandparent (*ggp*) and it also is connected to its previous grandparent (*gp*) by case 2. Through the other cases from 3 to 5, indirect connections are provided to the initiator. From case 3, the initiator can reach its *gp* through a sibling of the faulty node that has a lower height than the initiator. From case 4, it reachs its *gp* through a neighbor that has the same *gp* (i.e., same height as the initiator). And from case 5, it reachs its *ggp* through a neighbor that has the *ggp* as its *p* (i.e., lower height).

All cases from 1 to 5 directly or indirectly provide the initiator a new connection to a lower height node (i.e., its *gp* or *ggp*). Thus, new paths made by the above five cases are loop free. (End of proof)

INP-ACKINP is an essential procedure for the reconfiguration that is repeatedly used in the more complex reconfiguration steps (i.e., *CNFCF* and *ICNYP*). Fig. 21 describes the different responses of a node based on its current state when receiving an *ACKINP*.

```
RecvACKINP(init, j, j.p, caseNUM) from j {
  if (init == myID)
  {
        myID.p = j;
        myID.gp = j.p;
        myID.flowdirection = LOW;
        Broadcast ACPINP (myID, myID.p, caseNum) to its neighbors;
  }
  else {
        if (state == recvICNYP )
              Send ACKICNYP (init, myID, j, j.p)
                    to the node that gave ICNYP;
        else if (state == recvCNFCF)
              Send ACKCNFCF(init, myID, j, j.p) to the node
                    that gave CNFCF;
  }
}
```

Fig. 21. Procedures according to the different states when *ACKINP* is received.

2) CNFCF

If the detecting node n_i cannot find a cycle free parent with one of above five cases within a given time, node n_i lets each sibling node know this fact by broadcasting a *CNFCF (init, myID, init.p, init.gp)* control packet. *CNFCF* stands for *Cannot Find Cycle Free*. Each sibling that hears the *CNFCF* starts to find its new parent by broadcasting an *INP* control packet. When a sibling receives *ACKINP* from its neighbor that satisfies one of the above five cases, it sends *ACKCNFCF (init, myID, finderID, finderID.p)* control packet to the initiator, node n_i. *ACKCNFCF* stands for *Acknowledgement of CNFCF* and *finderID* is the ID of a sibling's neighbor that gives *ACKINP* to the sibling. When a sibling does not receive *ACKINP*, it broadcasts *CNFCF*. If a sibling that is not in the transmission range of the detecting node n_i receives *CNFCF*, it also can help node n_i by checking its neighbors by sending *INP*.

If node n_i receives *ACKCNFCF* messages from some of its siblings, it chooses one of the siblings as its new parent and broadcasts an *ACPCNFCF (init, siblingID, finderID,*

finderID.p) control packet. *ACPCNFCF* stands for *Acceptance of CNFCF*. The *ACPCNFCF* message also helps other nodes (children or siblings of node n_i) to find their new parent or grandparent. Fig. 22 shows these procedures. By helping each other, the possibility for the detecting node n_i to find a new parent is increased and the siblings can find their new parent without much extra message traffic.

```
recvCNFCF(init, i, init.p, init.gp)
{
        myid.intersibling = i;
        state =recvCNFCF;
        Send INP (init, myID, init.p, init.gp);
        If (TO(ACKINP)) {
                state = sendCNFCF;
                send CNFCF(init, myID, init.p, init.gp) to myID.sibs except node i;
        }
}
```

Fig. 22. Procedures when *CNFCF* is received.

Fig. 23 shows a situation that node *A* can find its new parent through sibling *B* when its current parent *P* is dead. Node *B* finds a new parent *C* after receiving a *CNFCF* message and node *A* changes its new parent with node *B* after receiving a *ACKCNFCF* message from node *B*. If node *A* does not have sibling *B*'s help via a *CNFCF* message, it can find its new parent through an *UNKNOWN* node *H* by spending more effort, as explained below, but at the cost of additional energy consumption.

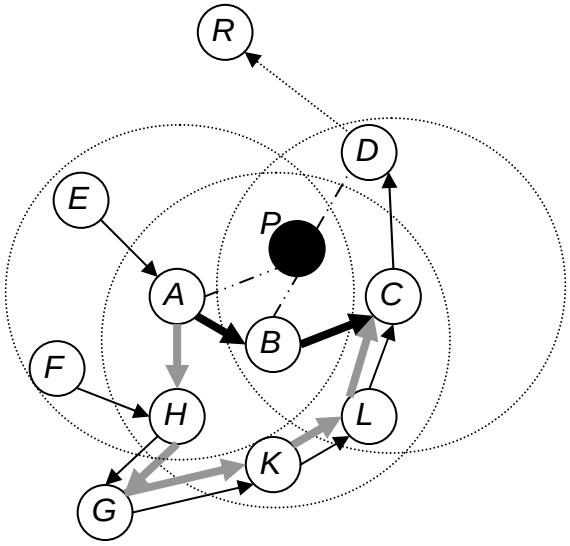


Fig. 23. When node A can have sibling B's help.

3) ICNYP

When the initiator node n_i still cannot find its *LOW* direction node through *INP* and *CNFCF* procedures, it broadcasts an *ICNYP (init, myID, init.p, init.gp)* control packet to each child to check if a child can find a *LOW* direction node. By broadcasting *INP*, each child that receives *ICNYP* tries to find a *LOW* direction node. *ICNYP* stands for *I Cannot Be Your Parent*. Fig. 24 shows these procedures.

```
recvICNYP(init, i, init.p, init.gp)
{
        state =recvICNYP;
        Send INP (init, myID, init.p, init.gp);
}
```

Fig. 24. Procedures when *ICNYP* is received.

When a child n_{i-1} receives *ACKINP* from a neighbor, it sends *ACKICNYP (init, myID, finderID, finderID.p,)* control packet to node n_i in response to the *ICNYP* message. *ACKICNYP* stands for *Acknowledgement of ICNYP*. The message contains the neighbor's *ID* in *finderID* and its parent in *finderID.p*. If node n_i receives *ACKICNYP* messages with new paths from some of its children, it chooses one of the children as its new parent, broadcasts an *ACPICNYP (init, childID, finderID, finderID.p)*. The *ACPICNYP* message also helps other nodes (children or siblings of node n_i) to find their new parent or grandparent. *ACPICNYP* stands for *Acceptance of ICNYP*, and *childID* for the node which sent *ACKICNYP* and chosen as a new parent. When a child n_{i-1} receives *ACPICNYP* from node n_i, it sets the neighbor that finds a *LOW* direction node as a new parent by sending *ACPINP*. Figs. 25 and 26 show those procedures.

```
Receive ACKICNYP (init, k, finderID, finderID.p) from k {
   If (init == myID) {
      If (state == sendICNYP) {
         myID.p = k;
         myID.gp = finderID;
         remove k from its children list
         Broadcast ACPICNYP(init, k, finderID, finderID.p) to it neighbors
         state = Complete;
      }
   }
   else if (init != myID) {
      if (state == recvICNYP)
         Send ACKICNYP(init, myID, finderID, finderID.p) to its parent;
   }
}
```

Fig. 25. Procedures when *ACKICNYP* is received.

```
Receive ACPICNYP (init, k, finderID, finderID.p) from k {
If (k == myID) {
        If ( state == recvICNYP) {
                myID.p = finderID;
                myID.gp = finderID.p;
                add initiator to its children list;
                Broadcast ACPICNYP to its neighbors;
                state = Complete;
        }
} else if (init != myID) {
        if (state == recvICNYP) {
                myID.gp = k;
                remove k from its sibling list;
                state = Complete;
        } else {
                if (finderID == myID) {
                        add k into my children list;
                } else if (k == myID.p) {
                        myID.gp = finderID;
                        add initiator into my sibling list;
                }
        }
   }
}
```

Fig. 26. Procedures when *ACPICNYP* is received.

In Fig. 27, node *A* sends a *ICNYP* message since it cannot find its new parent through *INP* and *CNFCF* procedures. Since node *E* has a neighbor *U* that has node *D* as its *gp* (i.e., case 4), it can choose node *E* as its new parent. Fig. 28 shows the message sequence for this situation.

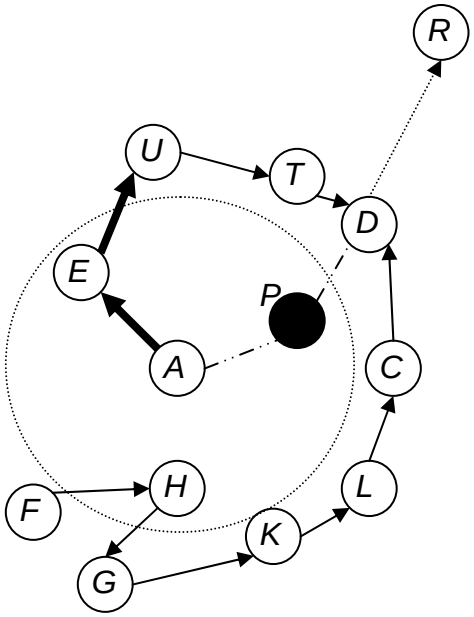


Fig. 27. When node A can find its new parent using *ICNYP*.

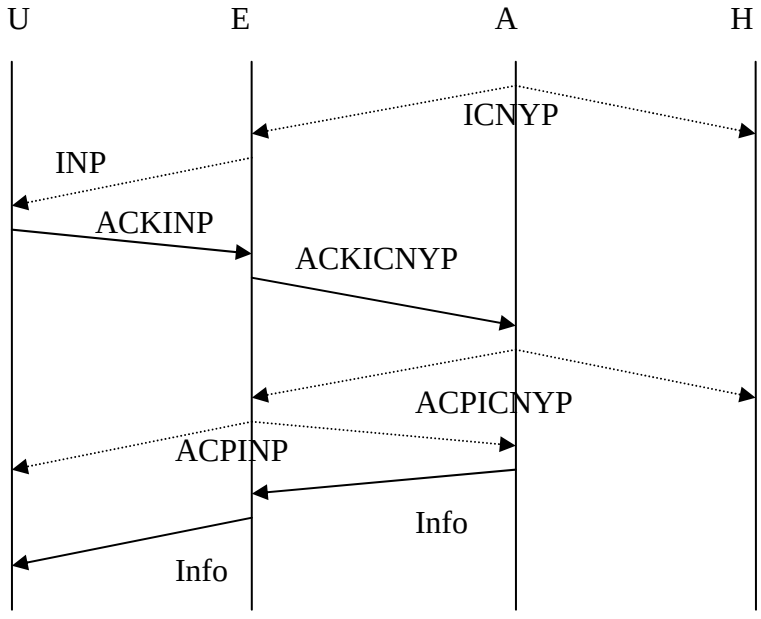


Fig. 28. Message sequence for Fig. 27.

4) INI and PFIND

When the initiator node n_i still cannot find its *LOW* direction node with *ICNYP* procedures, node n_i tries a node among *UNKNOWN* direction neighbors as its new (candidate) parent. Before doing this, node n_i gets more local information that helps it select an *UNKNOWN* node by broadcasting a *INI (myID)* control packet. Through *ACKINI (INIsender, myID, myID.p, myID.gp)* from the neighbors, node n_i receives that information. *INI* stands for *I Need Information* and *ACKINI* stands for *Acknowledgement of INI*, which carries information about the responding node's neighbors. Fig. 29 shows these procedures.

```
recvINI(INIsender)
{
        Send ACKINI (INIsender, myID, myID.p, myID.gp) to INI sender;
}
```

Fig. 29. Procedures when *INI* is received.

```
recvACKINI(INIsender, i, i.p, i.gp) from i
{
   if ((i.p == myID) || (i.gp == myID) || (i.p ∈ myID.siblings) ||
      (i.gp ∈ myID.children) || (i.gp ∈ myID.siblings))
         i.direction = HIGH;
   else if (i.p == myID.p)
         i.direction = SAME;
   else
         i.direction = UNKNOWN;
   if (i.direction == UNKNOWN) {
         state=sendPFIND;
         send PFIND(PFINDsender, PFINDsender.p, PFINDsender.gp, Relayer) to i;
  }
}
```

Fig. 30. Procedures when *ACKINI* is received.

By collecting this information node n_i may capture the local connectivity among neighbors and filter the *UNKNOWN* nodes that should not be chosen as new parent candidates. Fig. 30 shows these procedures. *HIGH* flow direction neighbors are discarded. Then for groups of neighbors that all share the same path, all but the highest node is discarded, to avoid redundant work. If node n_i still has neighbors that have an *UNKNOWN* flow direction, it will randomly choose one of these nodes and send a *PFIND (PFINDsender, PFINDsender.p, PFINDsender.gp, Relayer)* control packet to the node. *PFIND* stands for *Path Find*.

The *PFIND* message propagates up the neighbor's path towards the root, checking at each relay node whether it can provide a *LOW* direction node by satisfying one of five cases discussed above. If a *LOW* direction node is found at a relay node, the relay node responds to node n_i with *ACKPFIND (myID, PFINDsender, myID.p)* along the reverse path direction. *ACKPFIND* stands for *Acknowledgement of PFIND*. Then node n_i sets the *UNKNOWN* node that relays *ACKPFIND* as its new parent and broadcasts *ACPPFIND (Acceptance of PFIND)*. If node n_i does not receive a *ACKPFIND* message within a time limit, the node sends a *PFIND* message through a different *UNKNOWN* node. If there are no *UNKNOWN* nodes remaining, node n_i back offs on its time limit and restarts *INI* and *PFIND* procedures since the network topology may have changed during the previous procedures. Reconfiguration tries end when node n_i reaches the back off limit. Figs. 31 and 32 show these procedures.

```
recvPFIND (PFINDsender, PFINDsender.p, PFINDsender.gp, Relayer)
{
   if ((PFINDsender.gp == myID) || (PFINDsender.gp ∈ myID.chd) || (PFINDsender.gp ∈ myID.gp) ||
      (PFINDsender.gp ∈ myID.sibling) || (PFINDsender.gp == myID.p) || (myID == Representative))
         Send ACKPFIND (myID, PFINDsender, myID.p) to sender;
   else
         forward PFIND to its parent;
}
```

Fig. 31. Procedures when *PFIND* is received.

```
recvACKPFIND(myID, PFINDsender, myID.p) from k
{
   if (PFINDsender == myid) {
         myID.p = k;
         myID.gp = k.p;
         Broadcast ACPPFIND(myID, myID.p, myID.gp, caseNUM) to its neighbor;
   }
   else
         forward ACKPFIND to PFINDsender;
}
```

Fig. 32. Procedures when *ACKPFIND* is received.

Fig. 33 shows node *A* can find its new parent by using the *PFIND* procedure after trying all other previous procedures. An *UNKNOWN* node *H* can become a new parent of node *A*. This is because node *L* has node *D* as its *gp* (i.e., case 4). Fig. 34 shows the message sequence for this situation. Fig. 35 shows the overall reconfiguration algorithm flow chart.

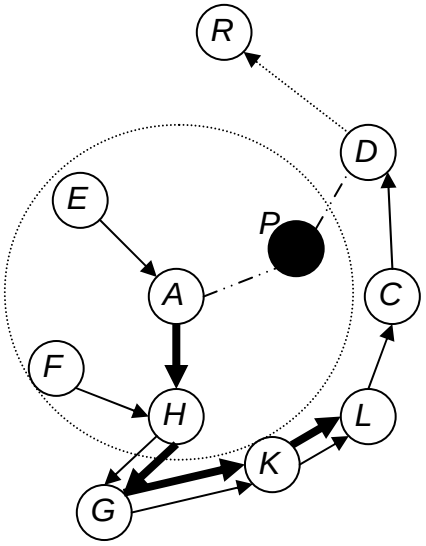


Fig. 33. When node A can find its new parent using *PFIND*.

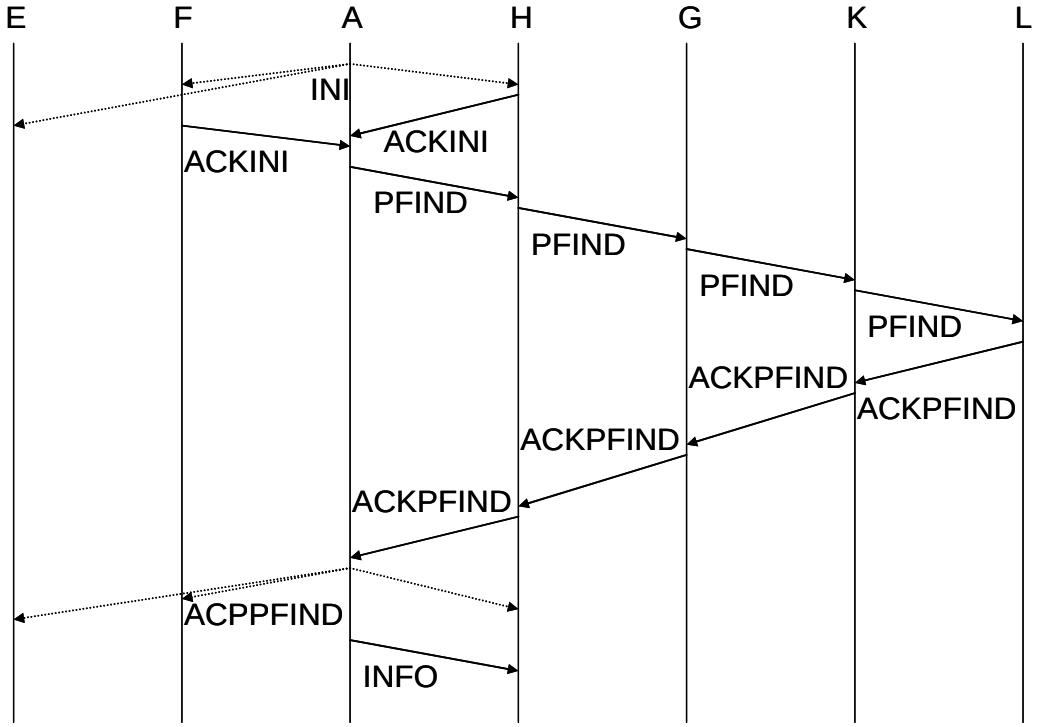


Fig. 34. Message sequence for Fig. 33.

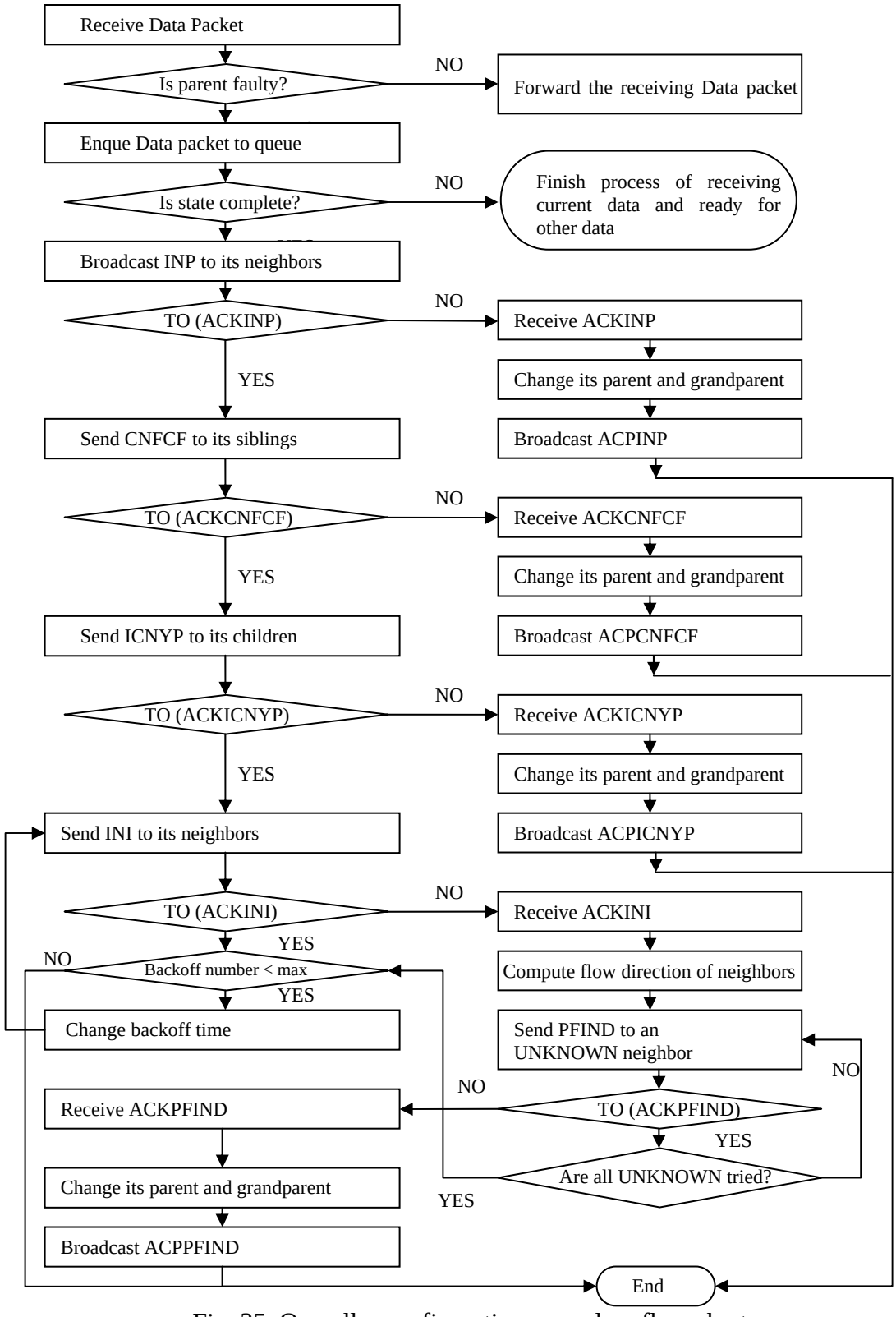


Fig. 35. Overall reconfiguration procedure flow chart.

Although the detecting node n_i has several sequential steps to find its new parent, these tries were bounded to save energy. It is better for a node to give up rather than consuming most of its energy searching for a new parent. There is an extended step to follow when a detecting node n_i can tolerate further energy consumption to find a new parent, such as when node n_i has really urgent data to deliver to the root. One solution is to flood the data. However, the node failure may mean that there is no way to create a path to root *R* in the current tree. Recognition of this fact is termed partition detection.

C. Partition handling

Node n_i and its descendants nodes might be partitioned from the current tree structure when node n_i's parent becomes faulty. It may be undesirable for the descendants of detecting node n_i to continue sending information to their parents on the current tree structure since this wastes energy. With the above local reconfiguration steps in *INP* algorithm, partition situations cannot be detected. By extending the existing reconfiguration steps to include a partition handling procedure, the nodes in the partition area can recognize the partition situation and stop sending data until the area rejoins the tree *T*.

The partition of node n_i and its descendant nodes does not mean that no path can be found from each node in the partition area to the root *R*. Each node may find a new path to root *R* by rebuilding a new communication tree structure or by flooding route request control packets that are used in reactive routing protocols like *AODV* [41].

The *INP* algorithm focuses on locally reconfiguring the existing tree communication structure against crash faults on a given network topology. Thus at most a few paths per

local reconfiguration situation are added to, removed from, and changed during reconfiguration. Thus the term “partitioned” in this dissertation means that node n_i and its descendant nodes could not find path to the root R using the partition detection procedure.

If the above steps of the *INP* algorithm fail, the partition detection procedure is initiated. Node n_i sends a *CHECK-PARTITION* control packet to its children. This happens after node n_i fails to receive a *ACKPFIND* message from any *UNKNOWN* neighbors. Each child again sends a *PFIND* message to its neighbors and waits for a *ACKPFIND* message. When a child receives *ACKPFIND* from a neighbor, it replies with *ACKCHECK-PARTITION (newparentID)* to node n_i with *newparentID* the neighbor’s ID. When a child does not receive any *ACKPFIND* within the given time limit, it sends a *CHECK-PARTITION* to its children. The same steps are repeated until each leaf node in the tree tries to find a new path.

When a leaf node cannot find a new path, it sends an *ACKCHECK-PARTITION* message with *newparentID = NULL* to its parent. When each parent node receives these messages from all of its children, it sends an *ACKCHECK-PARTITION* message with *newparentID = NULL* to its parent. When the initiator node n_i receives these messages from all of its children, partition has been detected. Fig. 36 shows the procedures.

```
Recv_CheckPartition()
{
    For each neighbor k except its parent, grandparent, children, and siblings
    {
        Send PFIND to k;
        If  TO(AckPFIND(k)
            NofNotFindNode++;
    }
    If (NofNotFindNode == Number of neighbors except its parent, grandparent, children, and siblings)
    {
        If (node is leaf)
            Send ACKCheckPartition(NULL, myid) to its parent;
        Else
        {
            For each child j
                Send CheckPartition() message to j;
        }
    }
}

Recv_ACKCheckPartition(p, from)
{
    If (p == NULL)
    {
        NofNotFindChild++;
        If ( (NofNotFindChild == Number of children)
        {
            If (recvCheckPartition)
                Send ACKCheckParition(NULL, myid) to its parent;
            else
                Detect partition;
        }
    }
    Else
    {
        if (recvCheckPartition)
            Send ACKCheckPartition(from, myid) to its parent;
        Else
        {
            Myid.Parent = from;
            Myid.gp = p;
            Send ACPICNYP(from);
            Return;
        }
    }
}
```

Fig. 36. Procedures of Check-Partition.

D. Joining spanning tree

When a node is newly deployed (reseeding) or restarts after a crash fault (e.g. battery

recharged), it must join the tree. When node n_i can reach a node n_j that belongs to the tree *T*, it joins the tree *T* by becoming a child of that node n_j. Since each node n_i does not have a parent and grandparent yet, it broadcasts *INP (init, myID, init.p, init.gp)* with *init.p=NULL* and *init.gp=NULL*. When any node n_j belonging to the tree *T* receives the *INP* message, it replies with *ACKINP (init, myID, myID.p, caseNUM)*. Then node n_i randomly selects one of the responding nodes as its new parent. Note that this approach can cause the spanning tree diameter to grow over time, but we are placing a priority on minimum-energy tree maintenance. If the diameter becomes too large, the spanning tree can be recreated.

In Fig. 37, there are two partition areas, I and II that are partitioned from the spanning tree *T* and each other. When nodes *O* and *P* are deployed, they cannot help those two partition areas to rejoin the tree. They just know that they cannot reach tree *T* since there is no *ACKINP* from a node belonging to the spanning tree *T*.

When nodes *S, T,* and *U* are additionally deployed, node *S* receives *ACKINP* from node *K* and sends *PARENT(S,K)* to its neighbors after sending *ACPINP (S,K,LOW)* to node *K*. Then the nodes in both partition areas I and II can rejoin tree *T* through node *S*. Node *O* functions as a connector between partition areas I and II.

As long as nodes are in a partition area, no procedure that makes parent-child relationships among the nodes is needed. When nodes are reconnected to the tree *T*, the relationships among the nodes are established. There is no merge procedure among partition areas themselves before those are reconnected to the tree *T*.

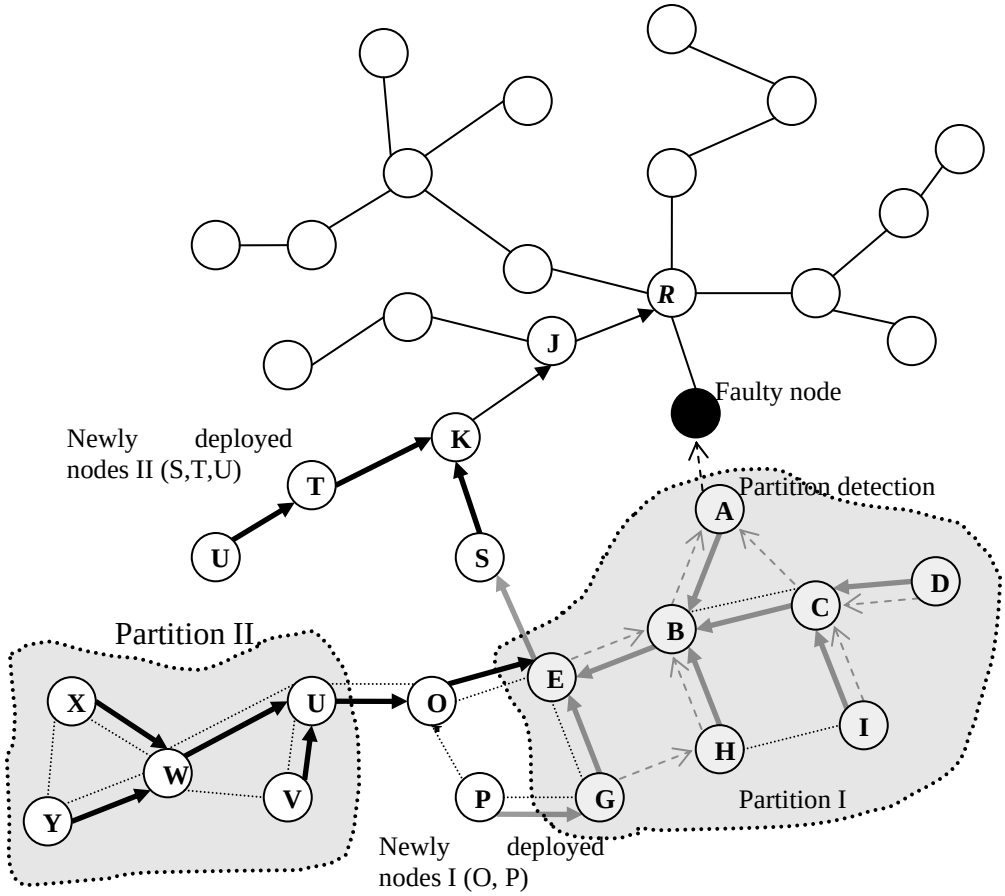


Fig. 37. Joining procedures.

IV. COMPUTATIONAL ANALYSIS

The *INP* algorithm was analyzed for energy consumption and message overhead. Since the energy consumed varies based on the particular network, the following subsections parameterize the network and compute the energy consumption and message overhead for specific functions and then for the entire algorithm.

Energy consumption is assumed not a function of distance but a function of message type. This is correct when the transmitter power is constant and transmitter energy is a function of message length. The energy to send a message of type X is denoted as $E(X_s)$. The energy to receive a message of type Y is denoted as $E(Y_r)$. Table I shows variables for computational analysis of *INP*.

TABLE I
VARIABLES FOR COMPUTATIONAL ANALYSIS OF *INP*

Variable	Description
N	total number of nodes (i.e., $N=\|F\|+\|FF\|$)
FF	set of all fault-free nodes
F	set of all faulty nodes
Nei[j]	total number of neighbors of node *j* including children and siblings
chd[j]	total number of children of node *j*
sibs[j]	total number of siblings of node *j*
unknown[j]	total number of *UNKNOWN* neighbors of node *j*
c	maximum number of neighbors
height[j]	height of node *j*

A. Establishing a tree

For establishing a tree, each node broadcasts its *PARENT* message and receives

PARENT messages from all fault-free neighbors. Since energy consumption for each fault-free node j is $E(parent_s) + Nei[j] \cdot E(parent_r)$, equation (1) is the energy consumption of all fault-free nodes.

$$\sum_{j=1}^{N} ((E(parent_s) + Nei[j] \cdot E(parent_r)) \text{ (where, } j \in FF) \quad (1)$$

(1) becomes (2) in the worst case when all nodes are fault-free with a maximum number of neighbors c.

$$N \cdot (E(parent_s) + c \cdot E(parent_r)) \quad (2)$$

The total number of messages exchanged does not exceed $c \cdot |FF|$ when the maximum number of neighbors is c, since each fault-free node sends a message. In the worst case when all nodes in the network are fault-free, the number becomes $c \cdot N$ or $O(N)$.

B. Local path reconfigurations

The following shows the energy consumption of path reconfiguration procedures.

1) INP

Energy consumption of an *INP* message sender node j when a neighbor that is not a child or sibling replies with *ACKINP* message is:

$$E(inp_s) + E(ackinp_r) + E(acpinp_s) \quad (3)$$

Maximum energy consumption of an *INP* message sender node j when all neighbors except children and siblings reply with *ACKINP* messages is:

$$E(inp_s) + (Nei[j]\text{-}chd[j]\text{-}sibs[j]) \cdot E(ackinp_r) + E(acpinp_s) \quad (4)$$

Energy consumption of an *ACKINP* message sender node is:

$$E(inp_r) + E(ackinp_s) + E(acpinp_r) \quad (5)$$

Energy consumption of other neighbor nodes is:

$$E(inp_r) + E(acpinp_r) \quad (6)$$

Thus the total energy needed for an *INP* procedure in worst case is expressed as (7). It shows when all neighbors except children and siblings reply with *ACKINP* messages.

$$(4) + (Nei[j]\text{-}chd[j]\text{-}sibs[j])\cdot(5) + (chd[j]+sibs[j])\cdot(6) \quad (7)$$

The total number of messages exchanged in worst case is *O(c)* with a maximum number of neighbors, *c.*

2) CNFCF

Energy consumption of a *CNFCF* message sender node *j* when a sibling replies with *ACKCNFCF* message is:

$$E(cnfcf_s) + E(ackcnfcf_r) + E(acpcnfcf_s) \quad (8)$$

Maximum energy consumption of a *CNFCF* message sender node *j* when all siblings reply with *ACKCNFCF* messages is:

$$E(cnfcf_s) + sibs[j]\cdot E(ackcnfcf_r) + E(acpcnfcf_s) \quad (9)$$

Energy consumption of an *ACKCNFCF* message sender node is:

$$E(cnfcf_r) + E(ackcnfcf_s) + E(acpcnfcf_r) \quad (10)$$

Energy consumption of other neighbor nodes except siblings is:

$$E(cnfcf_r) + E(acpcnfcf_r) \quad (11)$$

Thus the total energy needed for a *CNFCF* procedure in worst case is expressed in (12), when all siblings reply with *ACKCNFCF* messages.

$$(9) + sibs[j]\cdot(10) + (Nei[j]\text{-}sibs[j])\cdot(11) \quad (12)$$

The total number of messages exchanged in worst case is *O(c)* with a maximum

number of neighbors, *c*.

3) ICNYP

This procedure can be analyzed in two parts. One is related to the *ICNYP* message step and the other is related to the *INP* message step. Energy consumption of an *ICNYP* message sender node *j* when a child replies with *ACKICNYP* message is:

$$E(icnyp_s) + E(ackicnyp_r) + E(acpicnyp_s) \quad (13)$$

Maximum energy consumption of an *ICNYP* message sender node *j* when all its children reply with *ACKICNYP* message is:

$$E(icnyp_s) + chd[j] \cdot E(ackicnyp_r) + E(acpicnyp_s) \quad (14)$$

Maximum energy consumption of an *ACKICNYP* message sender node *k* that is a child of node *j* is:

$$E(icnyp_r) + E(ackicnyp_s) + E(acpicnyp_r) \quad (15)$$

This does not include the energy consumed by node *k* for the *INP* procedure before sending a *ACKICNYP* message to node *j*.

The energy consumption of all other neighbors of node *j* except its children is:

$$(Nei[j]-chd[j]) \cdot (E(icnyp_r) + E(acpicnyp_r)) \quad (16)$$

Thus, the total energy used for the *ICNYP* message step in an *ICNYP* procedure in the worst case is expressed as (17), when all children reply with *ACKICNYP* messages.

$$(14) + chd[j] \cdot (15) + (16) \quad (17)$$

In addition, each child *k* of node *j* executes an *INP* procedure that consumes the energy given in (7), with *k* used instead of *j*. Thus, the worst-case total energy used for the *INP* message step in an *ICNYP* procedure is expressed as (18).

$$chd[j]\cdot(7) \tag{18}$$

Finally, the total energy used for the *ICNYP* procedure in the worst case is expressed as (19).

$$(17) + (18) \tag{19}$$

The total number of messages exchanged in the worst case is $O(c^2)$, for a maximum number of neighbors, c.

4) INI and PFIND

Energy consumption of an *INI* message sender node j when *UNKNOWN* neighbors can reply with an *ACKINI* message is:

$$E(ini_s) + unknown[j]\cdot E(ackini_r) \tag{20}$$

When all neighbors except children and siblings are *UNKNOWN* (i.e., *Nei[j]-chd[j]-sibs[j])*, energy consumption of an *INI* message sender node j is maximized. Energy consumption of an *ACKINI* message sender node is:

$$E(ini_r) + E(ackini_s) \tag{21}$$

Energy consumption of other neighbor nodes is:

$$E(ini_r) \tag{22}$$

Thus, the total energy needed for an *INI* procedure is expressed as (23), when all *UNKNOWN* neighbors send *ACKINI* messages.

$$(20) + unknown[j]\cdot(21) + (Nei[j]-unknown[j])\cdot(22) \tag{23}$$

The total number of messages exchanged in the worst case is $O(c)$ or $O(1)$, with a maximum number of neighbors, c.

After receiving *INI* message(s), node j sends a *PFIND* message. Energy consumption

of node *j* when it tries the first *UNKNOWN* neighbor and finds a new path is:

$$E(pfind_s) + E(ackpfind_r) + E(acppfind_s) \quad (24)$$

The node *k* that confirms a cycle free path and replies with an *ACKPFIND* message back to the *PFIND* sender consumes the energy shown in (25).

$$E(pfind_r) + E(ackpfind_s) \quad (25)$$

Each message relay node *p* from node *j* to node *k* or vice versa consumes the energy shown in (26).

$$E(pfind_r) + E(pfind_s) + E(ackpfind_r) + E(ackpfind_s) \quad (26)$$

The number of relay nodes is $|height[k] - height[j]| - 1$. Thus, the total energy consumption of relay nodes is:

$$(|height[k] - height[j]| - 1) \cdot (26) \quad (27)$$

In the worst case, node *j* must try all its *UNKNOWN* neighbors. Equation (28) shows the energy consumption of node *j* in that case.

$$unknown[j] \cdot E(pfind_s) + E(ackpfind_r) + E(acppfind_s) <or> unknown[j] \cdot E(pfind_s) \quad (28)$$

Equation (28) comes from the fact that node *j* may have a new parent after trying all the *UNKNOWN* neighbors or may not have a new parent. That is, there is no such node *k* that confirms a cycle free path and replies with an *ACKPFIND* message back to the *PFIND* sender or a node *k* in the last *UNKNOWN* try. Thus, energy consumption in the final destinations of the *PFIND* message is:

$$unknown[j] \cdot E(pfind_r) <or> (unknown[j]-1) \cdot E(pfind_r) + (25) \quad (29)$$

The first case is for when *PFIND* sender *j* receives the *PFIND* message that it sent and the second case is for when node *k* consumes the energy shown in (25) after other

previous *PFIND* messages have arrived at node *j*.

For each *UNKNOWN* node try, the total energy consumption of relay nodes (27) is used. Thus (30) is used to get the total relay node energy consumption.

$$unknown[j] \cdot (27) \tag{30}$$

Therefore, the total energy consumption of the *PFIND* procedure is (31).

$$(28) + (29) + (30) \tag{31}$$

The total number of messages exchanged during *PFIND* is $O(unknown[j] \cdot height[root])$ in the worst case. Since the number of unknown nodes cannot exceed the maximum number of neighbors, the total number of messages is bounded by $c \cdot O(height[root])$. Height can range from $N-1$ to $O(log(N))$.

C. Competitive analysis

INP is evaluated with single path with repair routing (*SWR*) [18] since the reconfiguration of a single path is a common problem to both algorithms and *SWR* uses local path repair similar to *INP*.

The energy consumption of the algorithms is evaluated by using the per-packet analysis method [42]. Per-packet energy consumption is computed as follows:

$$Energy = C + (Power/DataRate) \times packet\ size \tag{32}$$

where *C* is the constant overhead per packet, *Power* is the transmitter or receiver power, *DataRate* is the data rate of the channel after removing encoding overhead, and *packet size* is the length of the packet. Power per bit is usually higher for transmitting than receiving [43]. The constant *C* accounts for per-packet computation, carrier sensing, sending or receiving any coordination packets such as RTS/CTS, and transceiver

wakeup time. The value of *C* depends on whether the traffic is broadcast or unicast: *C(B)* or *C(U)*. Unicast has higher energy consumption since it has an acknowledgement packet. After the constant amount *C* of energy consumption, each node consumes energy for receiving or for sending data packets, depending on the packet length. A radio transceiver model that uses the same power for receiving and listening is assumed [43]. Table II shows the parameters used in the analysis, taken from [43]. Transmit power is assumed to be constant, independent of neighbor location within the transmission range.

TABLE II
PARAMETERS FOR COMPUTATIONAL ANALYSIS OF *INP* AND *SWR*

Parameter	Value
Transmit power	24.75 mW
Receive power	13.5 mW
Bandwidth	20 kbps
MAC	S-MAC
Duration of periodic listen	115 ms
Duty cycle	10%
Number of nodes	1024

The calculation results below show the energy consumption for both algorithms for reconfiguration of one faulty node, both when a single message is delivered, and when all of the children of the faulty node deliver messages.

The following assumed topologies are reasonable since the characteristics of both algorithms are reflected on the results obtained by using those topologies. For example, each sibling of the detecting node can find its new parent without its own effort in the

INP algorithm unlike in the *SWR* algorithm.

Although each node in the tree can have different number of children in actuality, an assumption that each node has *r* number of children was used for the following calculations. Height of the tree is $O(\log_r N)$.

Fig. 38 shows the energy consumption for different numbers of children of the detecting node. The detecting node has 10 neighbors (including children and siblings, but not including the faulty parent). There are no siblings assumed in Fig. 38. Fig. 39 shows energy consumption for different number of sibling neighbors of the detecting node. The detecting node has 10 neighbors and 5 children. Note that a sibling might not be a neighbor if it is out of transmission range, but that is not considered in these examples.

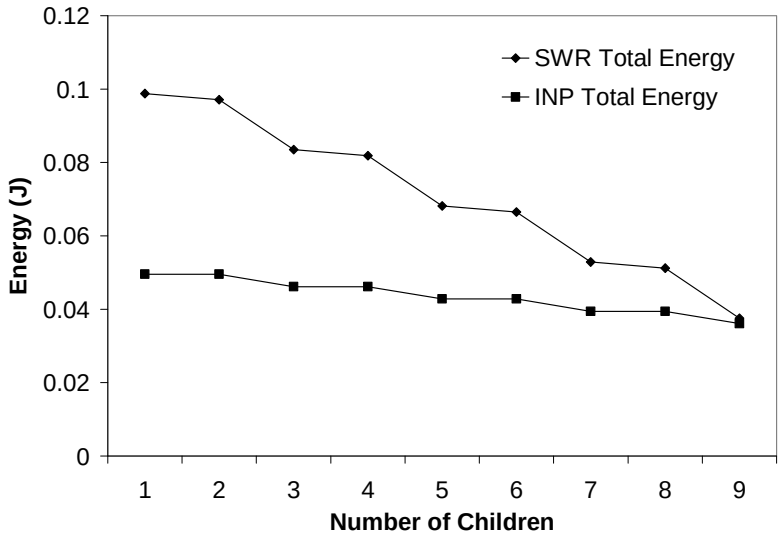


Fig. 38. When neighbor(s) can give direct response to *INP* or *HREQ* under different number of children.

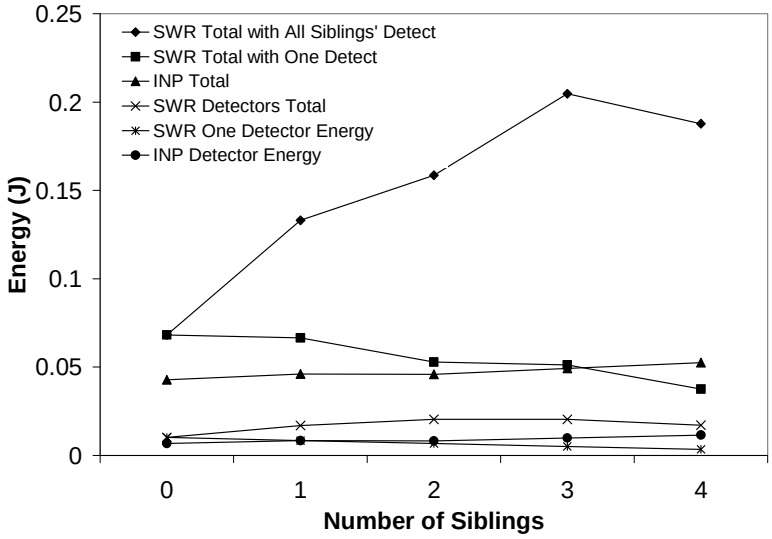


Fig. 39. When neighbor(s) can give direct response to *INP* or *HREQ* under different number of siblings.

In Figs. 38 and 39, half the neighbors of the detecting node that are not children or siblings are assumed to be able to directly guarantee a loop free path through them (cases 1 to 5 in *INP*) and reply with an acknowledgement (*ACKINP* in INP, *HREP* in *SWR*). For *SWR*, the *TTL* field in the *HREQ* message is set to 3, as in [18]. This limits the length of the route the *HREQ* message can travel [18]. Each neighbor that has a higher hop number than the value of the detector relays the *HREQ* message up until *TTL* becomes 0. In both algorithms, children do not send *ACKINP* or *HREP* messages. When there are few children among the neighbors, more neighbors relay the *HREQ* message. For example, with 2 children, 4 neighbors relay *HREQ* since 4 other neighbors can directly send a *HREP* message. This causes the total energy used by *SWR* to fall with increasing number of children of the detecting node, as shown in Fig. 38. When there are

9 children (all neighboring nodes are children, except for the new parent), the energy costs of *SWR* and *INP* are the same. In *INP*, the number of children does not significantly affect the energy consumption, since the neighbors (including children) that cannot find any of the five cases just wait until the detecting node finds a new parent (*ACPINP*) or starts the next procedure (e.g. *CNFCF*).

Figs. 38 and 39 do not consider the amount of energy that *SWR* needs for updating node cost (height) values. These updates are postponed until a downward message is delivered to each child [18]. Outdated height values can result in loop formation during reconfiguration, requiring extra energy to detect and remove. In the *INP* algorithm, the neighborhood relationships are kept updated during reconfiguration, so loops can never occur during reconfiguration.

In Fig. 39, as the number of siblings of the detecting node rises, the *INP* algorithm uses more energy while *SWR* uses less energy. This is because siblings are not involved in the *SWR* repair procedure. The *HREQ* message is discarded by all siblings. In the *INP* algorithm, although siblings do not try to find a new parent, they are aware that their parent is dead by receiving an *INP* message, and make the detecting node their new parent. This causes the total energy to rise with the number of siblings, but completes the reconfiguration for all siblings.

In Fig. 39, "*SWR* Total with One Detect" is the total energy consumed by all nodes for the first detecting node to reconfigure. It falls as the number of siblings rises, since there are fewer non-sibling, non-children neighbors to reply to messages. The curve "*SWR* Total with All Siblings' Detect" is the total energy consumed by all nodes for all siblings

to detect the faulty parent and reconfigure. This is more realistic since eventually most nodes will communicate with the root. The curve "*SWR* One Detector Energy" is the energy expended by one child of the faulty node to detect and reconfigure. The curve "*SWR* Detectors Total" is the total energy consumed by all the siblings as they all detect the faulty parent and reconfigure.

The top curve in Fig. 39 rises with the number of siblings, due to the repeated reconfiguration process. As the number of siblings increases, the energy consumption starts to fall, since there are fewer non-child, non-sibling neighbors, so there is less communication.

Fig. 40 shows the energy consumption when the detecting node can find its new parent after a *CNFCF* message is used in the *INP* algorithm. Each node has 10 neighbors and 5 children. Each sibling is assumed able to give an *ACKCNFCF* message to the detector. Since each sibling tries to find its new parent via the *INP* procedure, the amount of energy that sibling(s) and their neighbors consume increases with the number of siblings. The "Detector's Neighbor's Total" is the total energy used by the detecting node's neighbors to receive messages from the detector. This does not change since the number of neighbors is fixed here. The "Detector's Total Energy" increases slightly with increasing number of *ACKCNFCF* messages received. As mentioned above, *SWR* does not use any sibling help in reconfiguration so cannot reconfigure in cases where *INP* uses the *CNFCF* message procedure.

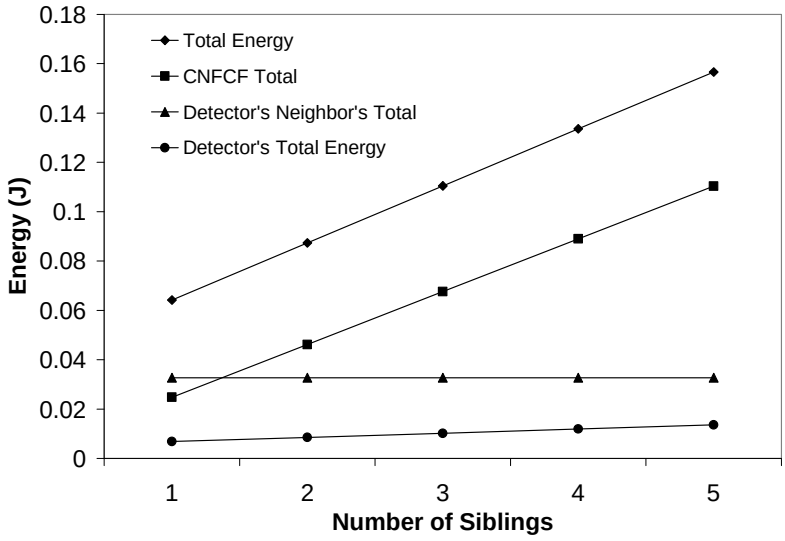


Fig. 40. When detector needs siblings' help without having direct response from its other neighbors.

Fig. 41 shows the energy consumption when the detecting node must use the *ICNYP* procedure to find a new parent. The detecting node has 2 siblings and each node including the detecting node has 10 neighbors. Each child is assumed able to give an *ACKICNYP* message to the detector. Since each child tries to first find its new parent via the *INP* procedure, the amount of energy that children and their neighbors' consume increases with the number of children. The detecting node's energy increases negligibly when the number of *ACKICNYP* messages received increase. Total energy consumption for the *CNFCF* procedure is not affected by the number of children since the number of detecting node's siblings involved in this procedure is fixed. As *SWR* does not use any children's help in reconfiguration, it cannot reconfigure in cases where *INP* uses the *ICNYP* procedure.

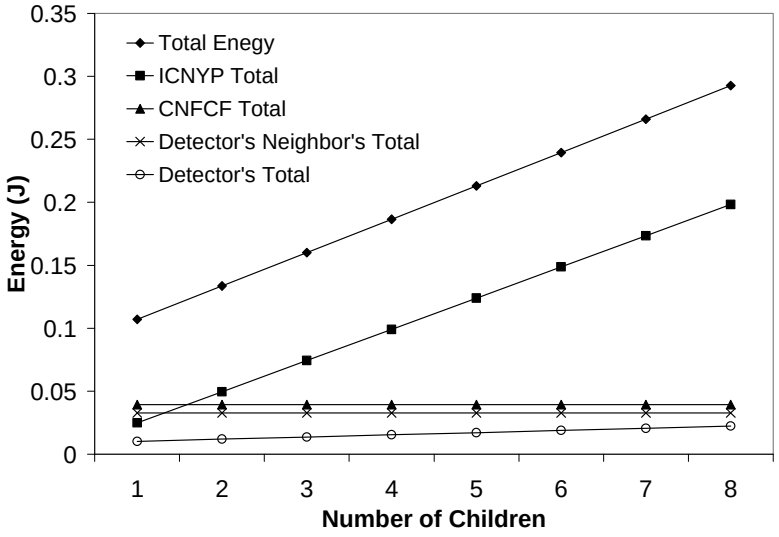


Fig. 41. When detector needs children' help without having siblings help.

Fig. 42 shows the energy consumption when the detector can find its new parent after trying all *UNKNOWN* neighbors in the *INP* algorithm. Each node has 10 neighbors. The detecting node has 2 siblings. The number of *UNKNOWN* neighbors is taken after subtracting children and siblings from the 10 neighbors. After the *INI* procedure, half of *UNKNOWN* neighbors are assumed as *HIGH* neighbors and disregarded in being considered as a new parent.

Among the remaining *UNKNOWN* neighbors, only one neighbor is assumed to receive an *ACKPFIND* message initiated from a child of root *R* along the reverse path direction after other failed *UNKNOWN* neighbors' tries. It happens when the *PFIND* message is assumed to be delivered to a child of root *R* without having a chance to find a new path, by assuming all five simple cases fail at all intermediate nodes on the path in the last *UNKNOWN* node's try. Previous failed tries occur when a *PFIND* message is sent to one

of detecting node's descendents. To measure the energy consumption of each failed try, the *PFIND* message is also assumed to travel as the length from the last *UNKNOWN* node to a child of root *R*.

UNKNOWN neighbors are assumed to be located at leaves of the tree for the worst case situation. With a network size 1024, each *UNKNOWN* neighbor's height is changed according to the different number of children. When the number of children is 2, the height is 9 (three children imply a height of six, etc.). Thus the total energy for the *PFIND* procedure falls when the number of children increases. As can be seen, the energy consumption for the *PFIND* procedure is higher than the energy consumption for other procedures.

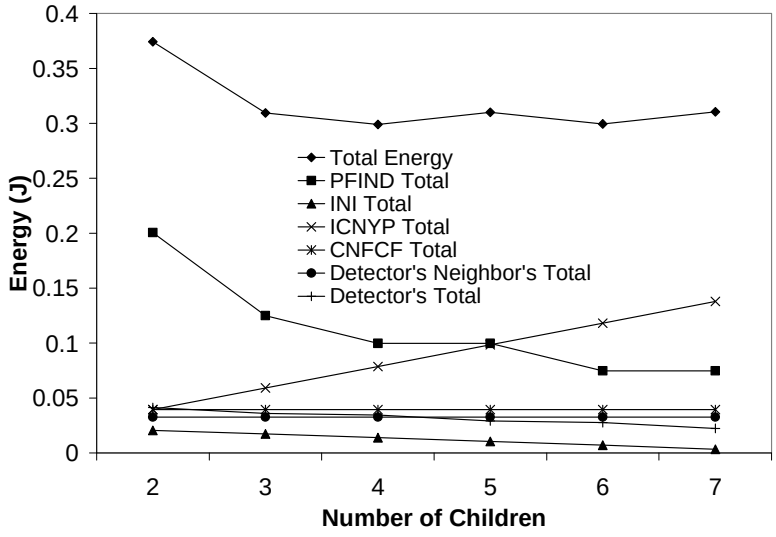


Fig. 42. When detector needs to try *UNKNOWN* neighbor(s) without having children's help.

V. SIMULATION ANALYSIS

INP and *SWR* are also evaluated through simulations and those algorithms are also compared with GRAdient broadcast [19] with fixed transmission range (*GRAB-F*) that uses dynamically made interleaving multiple paths. *GRAB* [19] was previously compared with *SWR* in [18] to evaluate the effectiveness of *SWR*. Three metrics that are used for comparing the performance of these approaches are average message delivery ratio, average information latency, and average energy consumption per data delivery. Information latency is defined as the amount of time spent for delivering a message from a source node to the root node R. These metrics are considered for different node densities and node failure rates.

A. Simulation environment

For simulations, we use NRLsensorsim [44], which extends the ns-2 network simulator [45] to facilitate simulating sensor networks. For simulating sensor networks that detect phenomena such as seismic activity or sound, NRLsensorsim adds the phenomenon notion to ns-2 [44]. This is implemented by a phenom “channel” attached to each node, separate from the regular data channel. Through the sensor agent that is attached to the phenom channel, each sensor node receives PHENOM packets that are broadcast by mobile PHENOM node(s) that are moving around along the paths provided by a PHENOM “routing” protocol. Nodes deliver the received PHENOM packet events to the node’s sensor application. Nodes react to the phenomenon (PHENOM packets) according to the function defined by its sensor application [44]. In this work, the nodes will transmit “phenomenon detected” messages to the root.

Table III shows the parameters that were used for the simulations. A radio transceiver model that uses the same power for receiving and listening is assumed, as in section IV [43].

TABLE III
PARAMETERS FOR SIMULATIONS OF *INP, SWR,* AND *GRAB-F*

Parameter	Value
Transmit power	24.75 mW
Receive power	13.5 mW
Idle listening power	13.5 mW
Sense power	0.00175 mW
Bandwidth	2Mbps
MAC	802.11b
Network size	400
Network type	Random
Maximum fault rate	20%
Transmission range	Fixed
Radio propagation model	Two-ray ground reflection
Antenna model	Omni directional
Average initial number of neighbors	7, 15

Nodes are randomly distributed on a plane. Fig. 43 shows 400 randomly deployed and located sensor nodes and a destination (sink) node on a fixed 2000 × 2000 m^2. All nodes are fixed except a mobile PHENOM node [44]. The destination node is located at (1067, 1909) and the PHENOM node [44] is initially located at (300, 300). A communication tree is constructed and the PHENOM node moves around inside the sensor field with a constant velocity (e.g., 500 m/s or 1000 m/s). When the phenomenon moves into the transmission range of a node, the node sends a “phenomenon detected” packet to the root node. The circles in Fig. 43 show one fixed transmission range occurred when nodes

send those packets. Although it is looked like many different transmission ranges depending on the sending nodes since the snapshot has various sizes of the circles, but it is all the same. Up to 20% of randomly selected nodes (i.e., 80 out of 400 in Fig. 43) become faulty at different times, five sets of faulty nodes are used without duplication and the results averaged for the simulations. The same conditions are also used in the *SWR* and *GRAB-F* simulations. One thing we want to emphasis is that *GRAB-F* uses a fixed "width" of the forwarding paths, in contrast to the *GRAB* algorithm, which assumes variable transmit power and so variable path width. Two different transmission ranges are used in the simulations, one with an average of 7 neighbors, and one with an average of 15 neighbors.

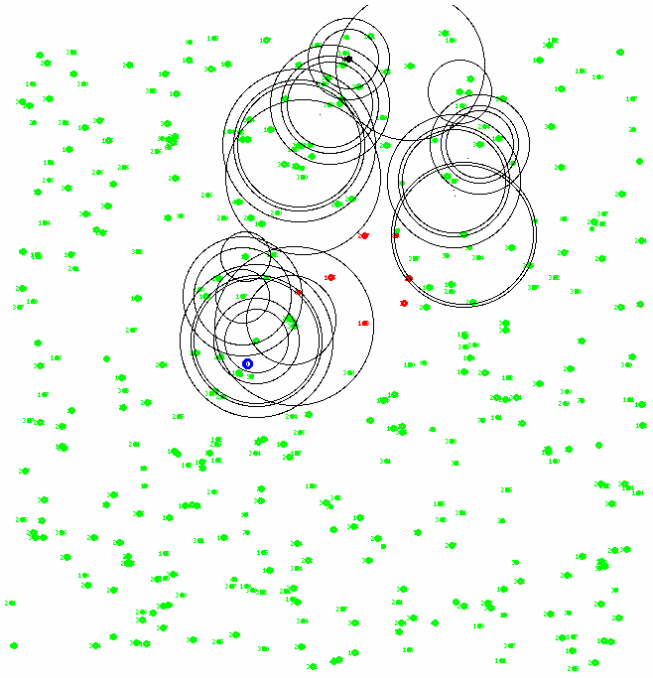


Fig. 43. Random distribution of 400 nodes.

B. Simulation results

Figs. 44 and 45 show average data delivery rates for 15 and 7 neighbors. In both cases, there is only a small difference (less than 3%) in delivery ratios between *INP* and *SWR* for the same number of faulty nodes. In a dense network, Fig. 44, *INP* has a smaller drop in delivery ratio with increasing faults (less than 1%) than *SWR* has (less than 3%). This is because *INP* could handle all reconfiguration situations that occurred, while *SWR* could not. Thus *INP* has a higher delivery ratio than *SWR* after 9 faults.

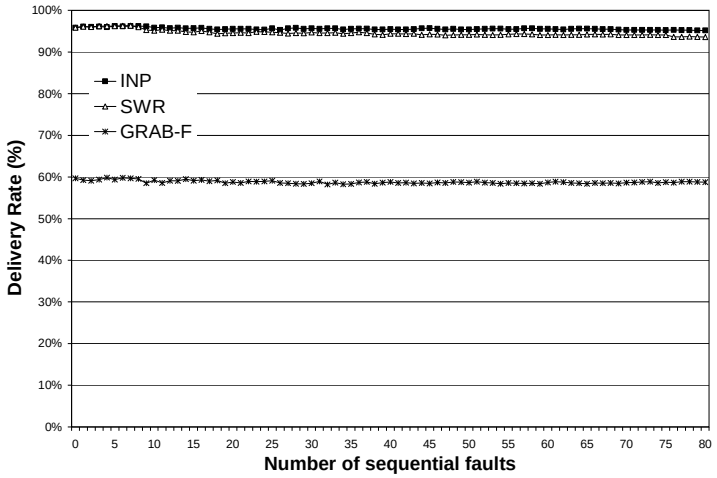


Fig. 44. Average delivery ratio for 15 initial neighbors.

In Fig. 45, *INP* has a higher delivery ratio than *SWR* after 33 faults. In a sparse network with increasing faults, the chance that nodes cannot find new parents increases in both approaches. Nevertheless, *INP* found more new parents than *SWR*. The single path delivery schemes (*INP*, *SWR*) always had a delivery ratio more than 25% higher than *GRAB-F*, a multiple path delivery scheme using broadcast.

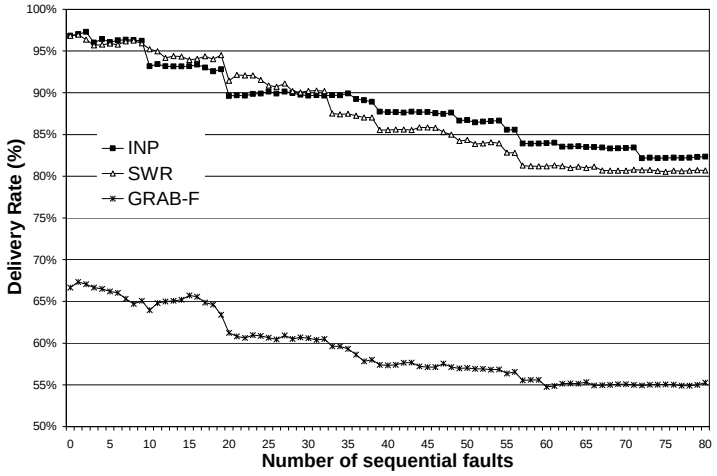


Fig. 45. Average delivery ratio for 7 initial neighbors.

Figs. 46 and 47 show the specific reasons for undelivered data messages in *INP* and *SWR*. "Undelivered messages" are the sum of "Holding messages" and "Dropped messages". "Holding messages" are the messages stored in a node queue because the node could not find a new path. "Dropped messages" are either the messages dropped at fault-free nodes or the messages dropped when nodes become faulty. In Fig. 46, there is little difference between *INP* and *SWR* for "Dropped messages" (less than 10 more messages dropped in *SWR*). There are no "Holding messages" for *INP* because all nodes find their new parents and thus "Undelivered messages" are all from "Dropped messages". *SWR* has many more "Holding messages" than *INP* starting from 9 faults. In Fig. 47, there are some sharp increases in undelivered message with increasing faults. Figs. 48 to 62 show the reasons why the undelivered messages are dropped or held.

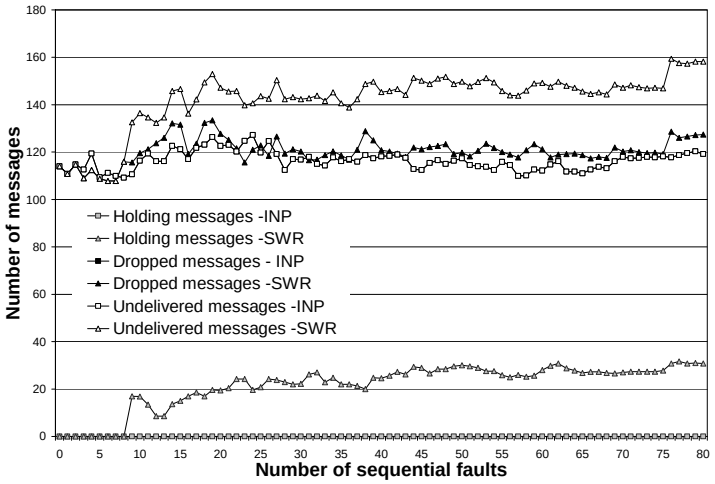


Fig. 46. Reasons for undelivered messages for 15 neighbors.

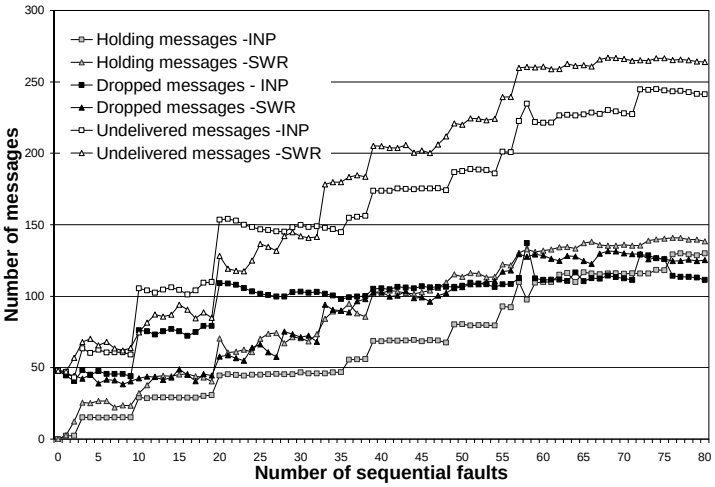


Fig. 47. Reasons for undelivered messages for 7 neighbors.

Figs. 48 and 49 show the total number of messages that were dropped by MAC collision for 15 and 7 neighbors. This includes data for the same message dropped repeatedly. In a dense network, Fig. 48, more collisions occur than in a sparse network, Fig. 49. The collisions decrease with increasing node failures, since fewer nodes are

sending messages.

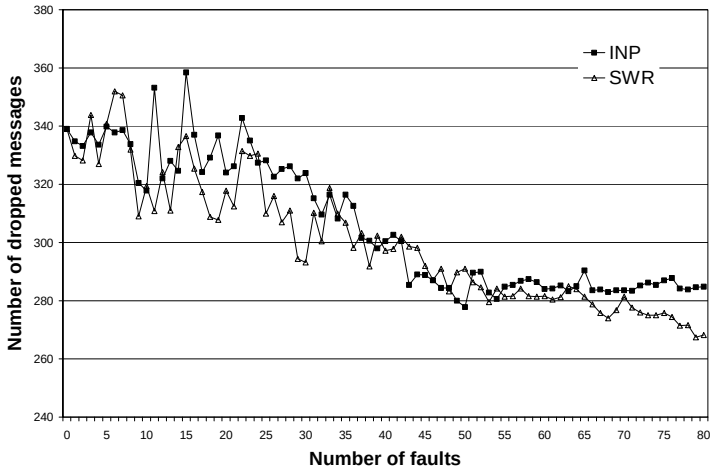


Fig. 48. Average messages dropped by Drop-MAC-Collision for 15 initial neighbors.

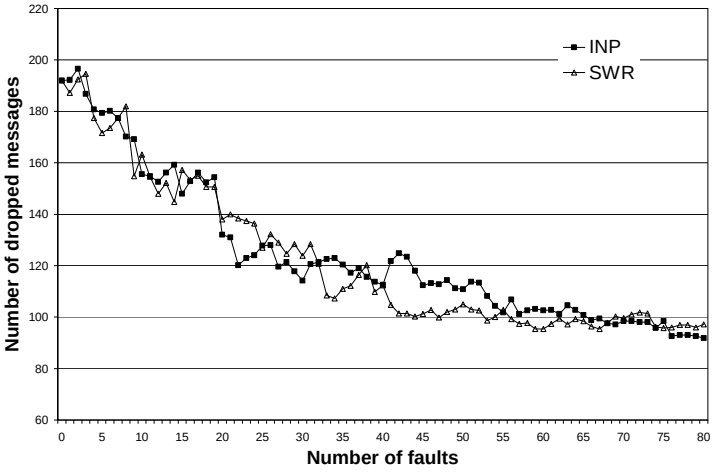


Fig. 49. Average messages dropped by Drop-MAC-Collision for 7 initial neighbors.

At the MAC layer, a message is retried up to 4 times before giving up. If the retry limit is reached, this route failure is reported to the RTR layer by the link layer. Then Drop-RTR-MAC-Callback for the message occurs [45]. The number of messages permanently

dropped by MAC collision after reaching the retry limit is shown in Figs. 50 and 51.

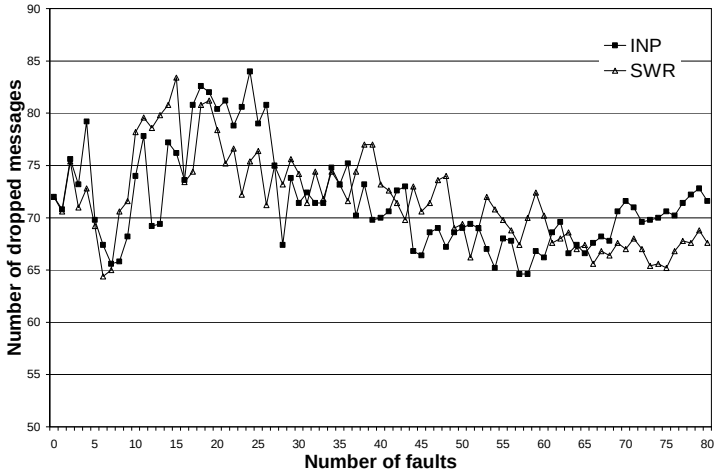


Fig. 50. Average messages dropped by Drop-MAC-Retry-Count-Exceed for 15 initial neighbors.

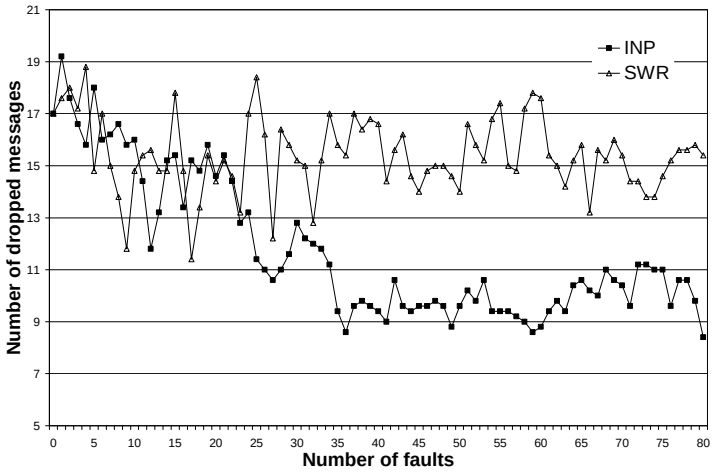


Fig. 51. Average messages dropped by Drop-MAC-Retry-Count-Exceed for 7 initial neighbors.

Figs. 52 and 53 show the messages dropped by Drop-RTR-MAC-Callback. These figures include the number of messages dropped by MAC-Retry-Count-Exceeded since

MAC-Retry-Count-Exceeded causes Drop-RTR-MAC-Callback.

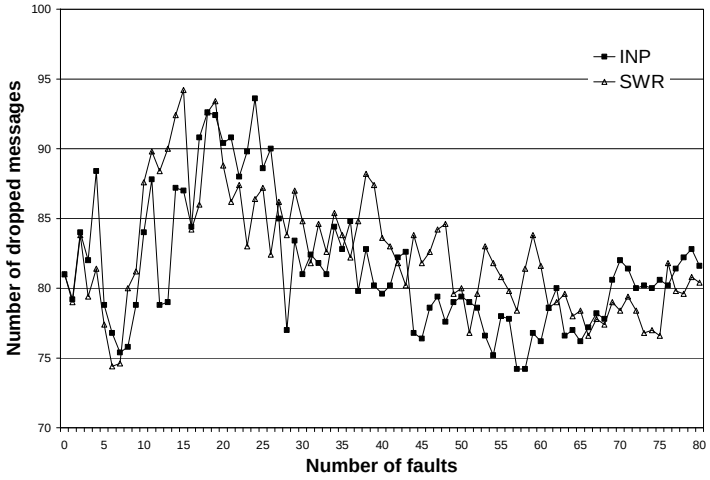


Fig. 52. Average messages dropped by Drop-RTR-MAC-Callback for 15 initial neighbors.

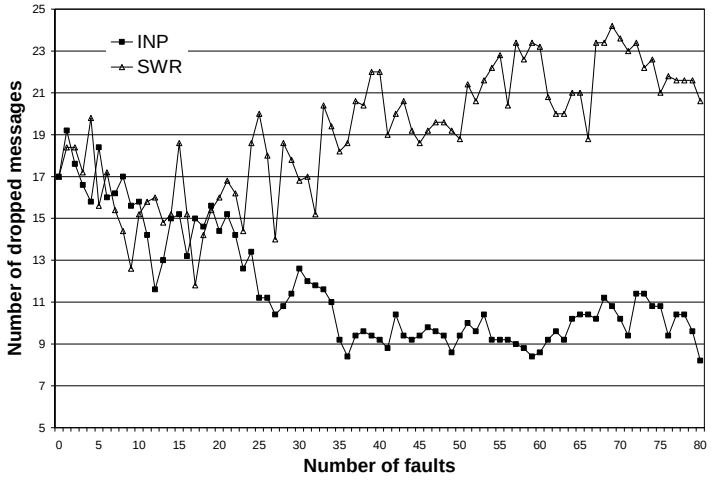


Fig. 53. Average messages dropped by Drop-RTR-MAC-Callback for 7 initial neighbors.

The maximum number of messages in the RTR queue is 64 (RTQ_MAX_LEN 64) and the maximum period of time that a message can stay in the queue is 30 seconds

(RTQ_TIMEOUT 30). Any messages reaching either limit are dropped. Fig. 54 shows the number of messages that were dropped by RTR queue full with 7 initial neighbors. Queue full drops do not occur with 15 initial neighbors. This is because most reconfigurations were successful and quickly performed in the dense environment. With 7 initial neighbors in Fig. 54, *INP* has many dropped messages and *SWR* has no dropped messages. This is because a forwarding node in *INP* keeps the received messages in the RTR queue, while the node in *SWR* sends the message back to the source when it cannot find its new parent, preventing a bottleneck in the queue. This explains why *SWR* has more held messages than *INP* in Figs. 46 and 47. It also explains why *INP* has more dropped messages than *SWR* before 53 faults in Fig 47.

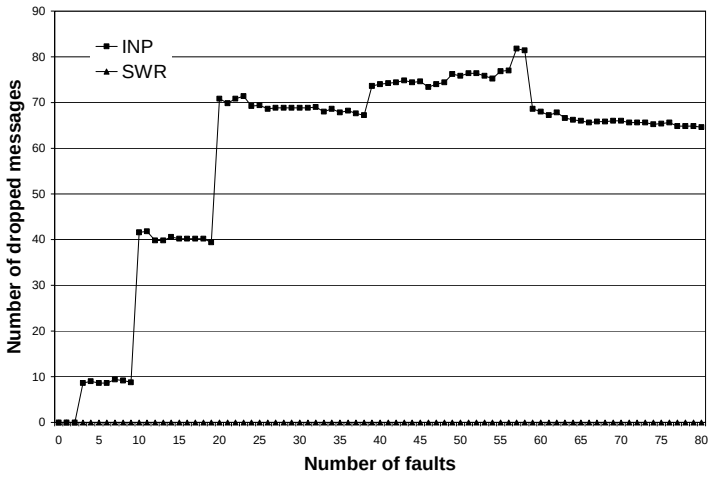


Fig. 54. Average messages dropped by Drop-RTR-Qfull for 7 initial neighbors.

In Figs. 55 and 56, Drop-RTR-Route-Loop occurs when a source node receives the messages it sent in both approaches. In *SWR*, Drop-RTR-Route-Loop also occurs when a

message forwarding node receives the message. We call this an inner loop. It is different from a loop that occurs when the source node receives the message that it sent. *INP* does not detect inner loops and messages are delivered until the time to live (TTL) limit is reached, then dropped. In *INP*, since a node that needs a new parent finds a *Low* direction node as its new parent, both kinds of loops are not expected. But in practice, loops and inner loops can occur when a broadcast control packet (e.g., *ACPICNYP*) is not delivered to neighboring nodes. An average of 0, 1.2, or 2.2 loops were found in Fig. 56 and an average of 0, 3.4, or 4.4 were found in Fig. 55. In *INP*, dropped packets caused by inner loops are not included in Figs 55 and 56 but included in Figs 57 and 58. In *SWR*, loops and inner loops are caused by not updating node costs (height). In a sparse network such as in Fig 56, there are many chances for these loops with increasing faults.

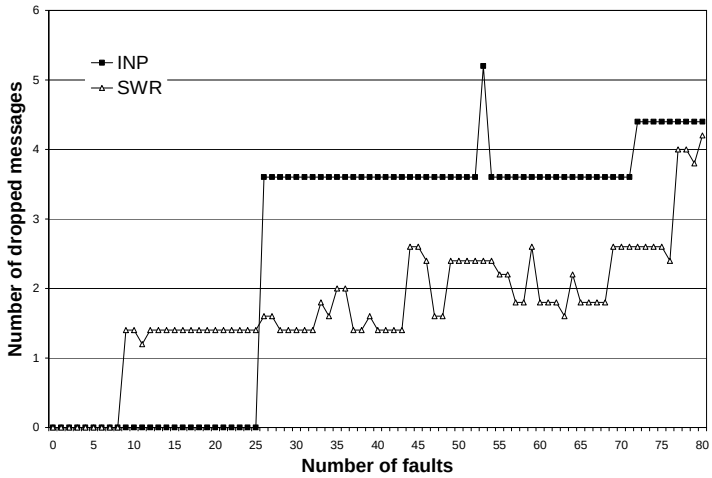


Fig. 55. Average messages dropped by Drop-RTR-Route-Loop for 15 initial neighbors.

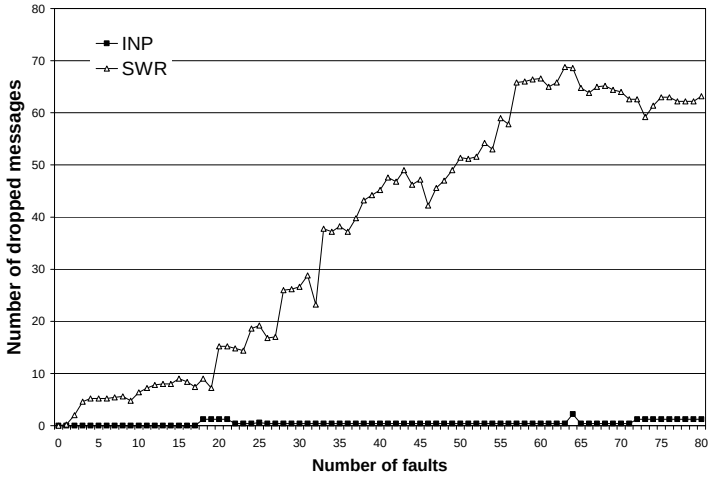


Fig. 56. Average messages dropped by Drop-RTR-Route-Loop for 7 initial neighbors.

In Figs. 57 and 58, Drop-RTR-TTL occurs when time to live (TTL) is exceeded. We set TTL to 32. In *INP*, TTL drops were also caused by inner loops, since *INP* does not provide inner loop detection.

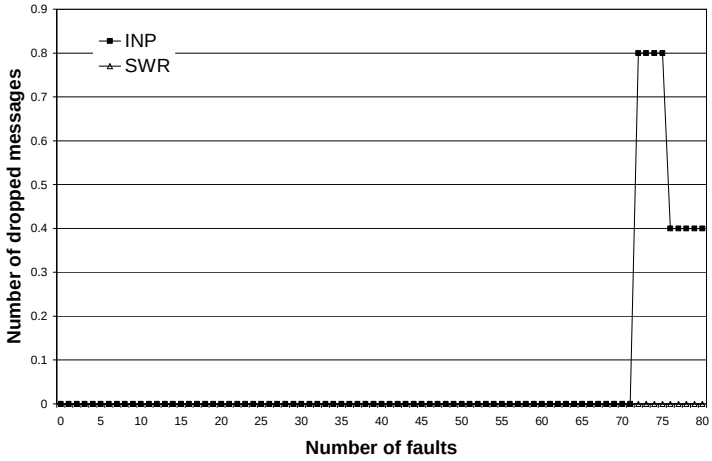


Fig. 57. Average messages dropped by Drop-RTR-TTL (32) for 15 initial neighbors.

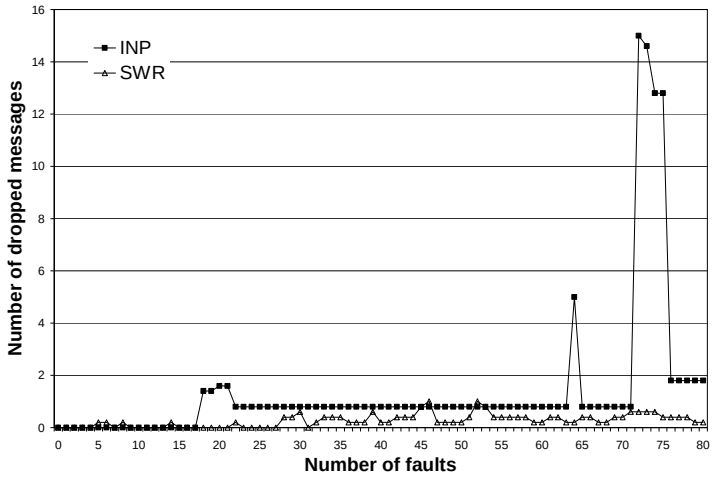


Fig. 58. Average messages dropped by Drop-RTR-TTL (32) for 7 initial neighbors.

Figures 59 and 60 show the average number of dropped messages among those buffered at the ARP table. DROP_IFQ_ARP_FULL happens only for unicast, not broadcast. The ARP module receives queries from the Link layer. If ARP has the hardware address for the destination, it writes it into the MAC header of the message. If not, it broadcasts an ARP query, and caches the message temporarily. There is a buffer for a single message for each unknown destination hardware address. When an additional message to the same destination is sent to ARP, the earlier buffered message is dropped [45].

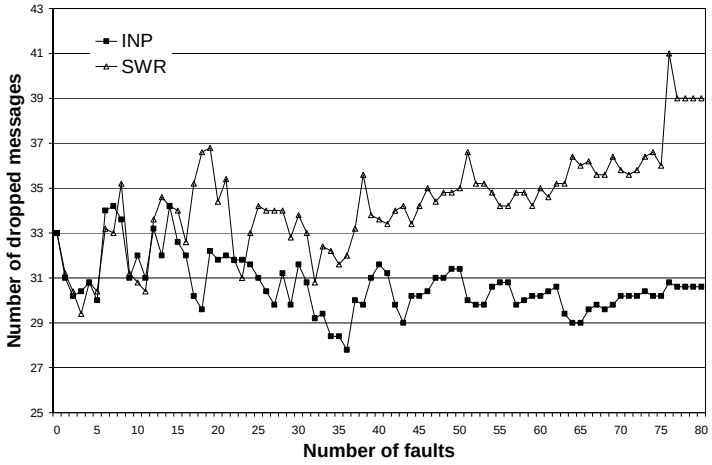


Fig. 59. Average messages dropped by Drop-IFQ-ARP-FULL for 15 initial neighbors.

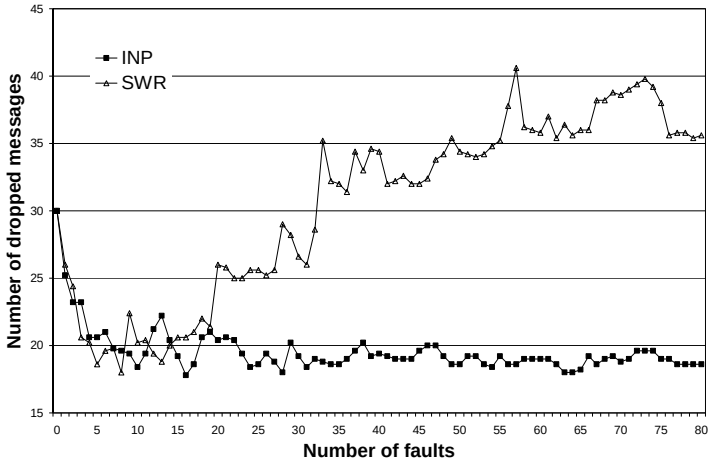


Fig. 60. Average messages dropped by Drop-IFQ-ARP-FULL for 7 initial neighbors.

In Figs. 61 and 62, Drop-End-of-Simulation occurs when a message remains in the interface queue at the end of simulation.

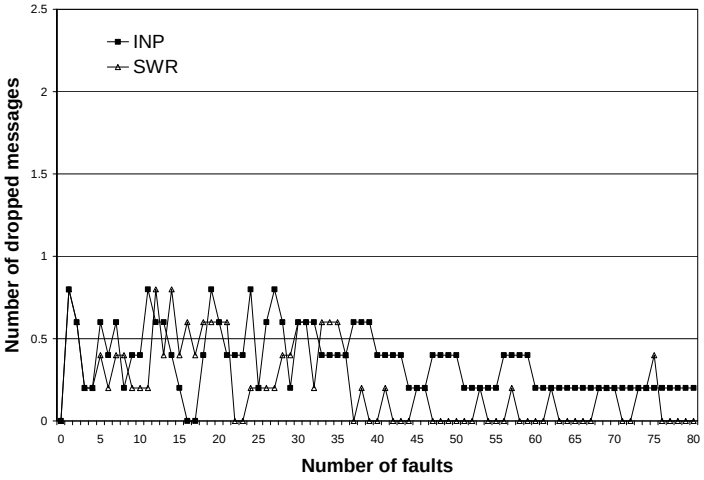


Fig. 61. Average messages dropped by Drop-End-of-Simulation at IFQ layer for 15 initial neighbors.

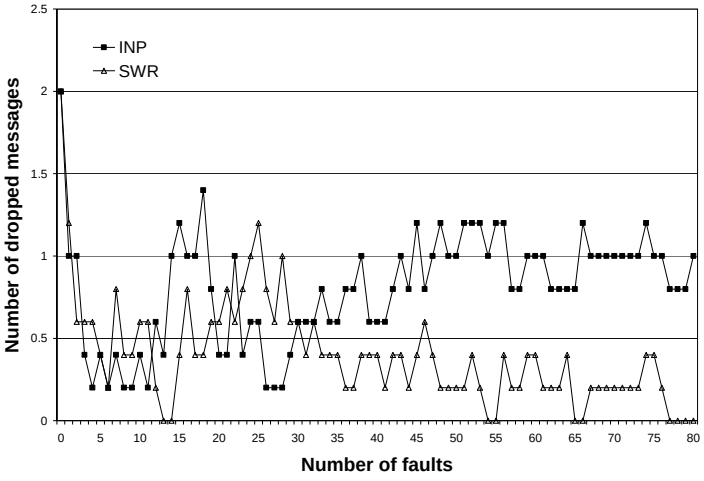


Fig. 62. Average messages dropped by Drop-End-of-Simulation at IFQ layer for 7 initial neighbors.

Figs. 63 and 64 show average data message latency. In a dense network, Fig. 63, with increasing number of faults, *INP* show less latency than *SWR*. This is because the five basic cases were used for most reconfigurations, and take little time. In a sparse network,

Fig. 64, *INP* has higher latency than *SWR*. This is because further procedures such as *PFIND* are used more often, and take longer time. Note that in these situations, *SWR* would drop the message.

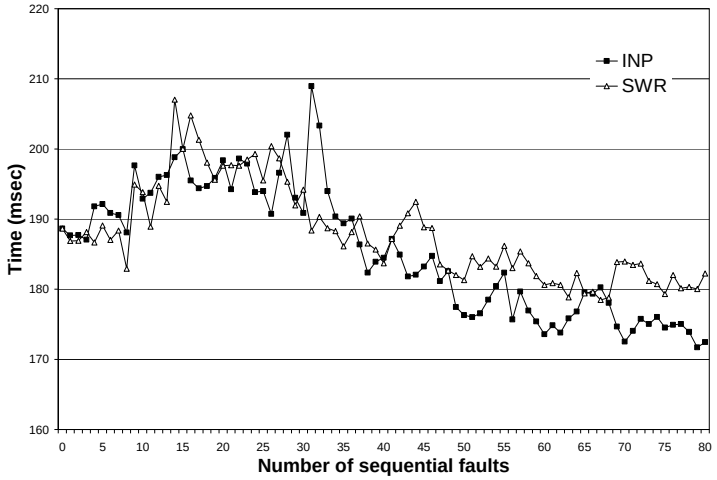


Fig. 63. Average latency for 15 initial neighbors.

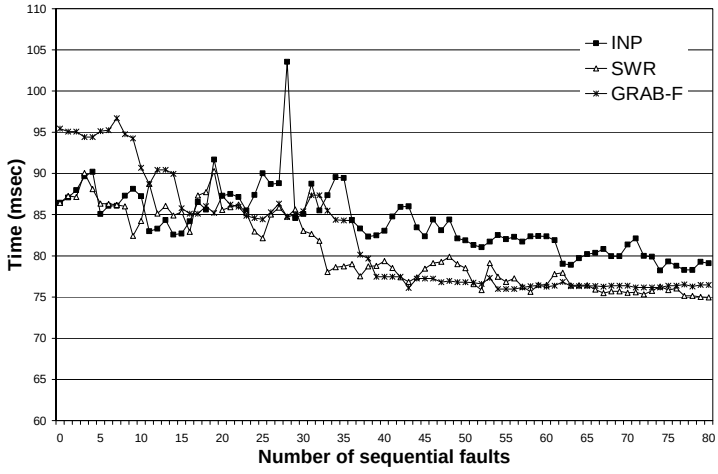


Fig. 64. Average latency for 7 initial neighbors.

Figs. 65 and 66 show average energy consumption per node. The energy for all activities in the simulation is included: making a tree, delivering data messages, sensing environment, and reconfiguring paths. The energy for making an initial tree and sensing the environment is the same in all three approaches. *INP* has lower energy consumption than *SWR* and *GRAB-F*, which is one of the primary goals of the research.

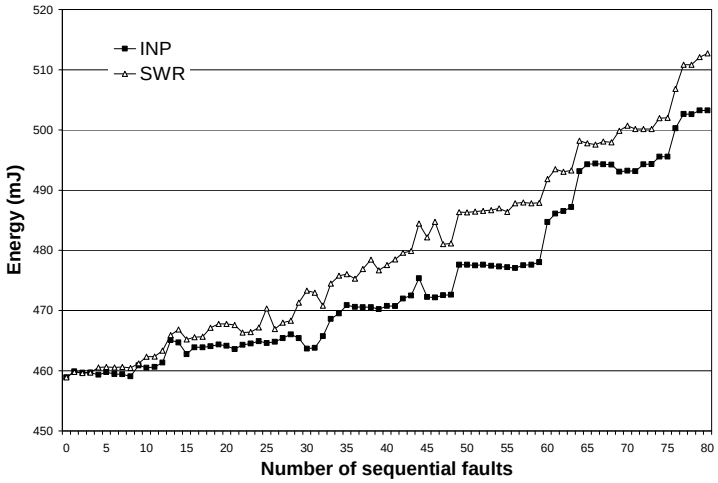


Fig. 65. Average energy per node for 15 initial neighbors.

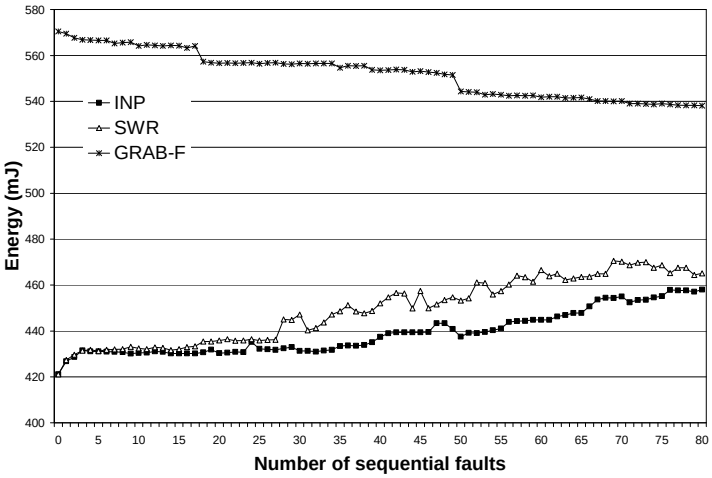


Fig. 66. Average energy per node for 7 initial neighbors.

Figs. 67 and 68 show average energy per node per successful message delivery. *INP* uses less energy than *SWR* or *GRAB-F*. In a dense network, Fig. 67, less energy is consumed per message than in a sparse network, Fig. 68, for both *INP* and *SWR*. This is because reconfigurations were easily done without consuming much energy.

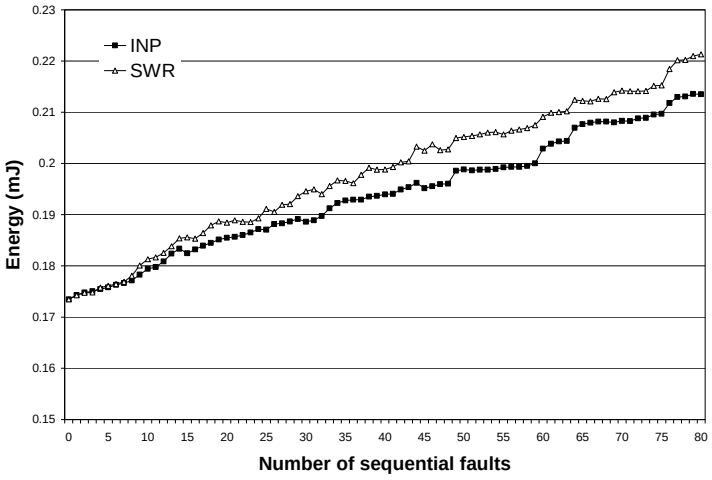


Fig. 67. Average node energy per message for 15 initial neighbors.

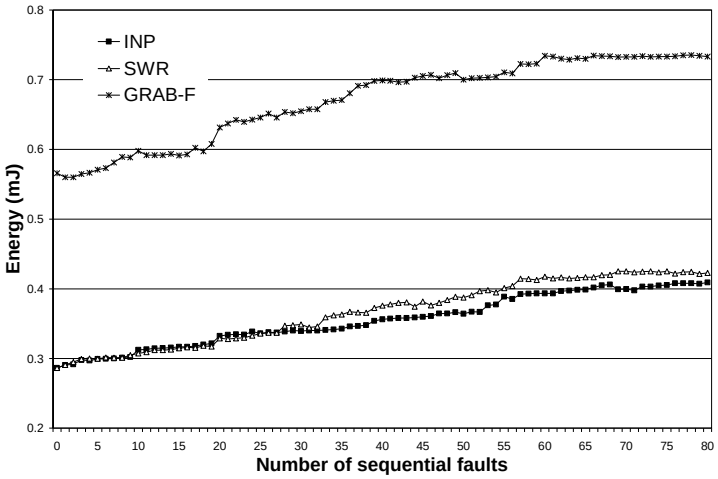


Fig. 68. Average node energy per message for 7 initial neighbors.

VI. SYSTEM LEVEL DIAGNOSIS ALGORITHMS FOR WIRELESS SENSOR NETWORKS

A. Introduction

Nodes in a sensor network can fail for many reasons, including battery depletion or destruction [3]. Failures include complete loss of a node, or internal faults that cause a node to operate incorrectly. Different mechanisms must be applied for diagnosing the different kinds (e.g., crash, malicious) of failures.

In this section, a new sensor-initiated [7] crash fault diagnosis algorithm is introduced for wireless sensor networks. Unlike monitoring/testing initiated by a request from an observer (i.e., observer-initiated approach [7]), sensors trigger the testing periodically or when they detect some signs/evidence of failure in a node. The problem with the observer-initiated approach is that the status of the sensor network cannot be known before the request from the observer/manager. This is not appropriate for sensor networks used in urgent and critical environments that need self-monitoring/testing mechanisms for a control observer to know constantly the fault/health status of sensors in the network. This proposed approach uses a one-to-one testing mechanism where each node is tested by one fault-free node by using unicast communications. It provides more reliable testing results than results produced by one-to-many testing that uses broadcast communication, with its accompanying problems (e.g., contention).

This approach uses a routing tree that is locally reconfigured in the face of broken paths due to faulty nodes, to report diagnosis information. By using dynamic reconfiguration, both static and dynamic faults are detected. And the newly

deployed/recharged nodes can join at any time in this algorithm.

Energy consumption and communication overhead in this approach increases with network size. To provide scalability, an extended approach is introduced, in which the network is partitioned into zones. Each zone has a local representative node and each node sends diagnosis information not to the network representative node (root), but to its local representative node. The local representative node sends summarized diagnosis information to the network representative node (e.g. number of node faults). Each local representative node knows each node's status in the local tree and the network representative node can recognize not the specific status of each node in the local tree, but the local tree's general network status. This reduces energy consumption and provides scalability.

The rest of the section is organized as follows. In subsection B, traditional system level diagnosis and the previously introduced algorithms are reviewed. Subsection C presents a new sensor-initiated fault diagnosis algorithm (*Repre*) and it is analytically compared with the *WSNDiag* algorithm. Subsection D presents a scalable distributed fault diagnosis algorithm (*Local*) and it is also analytically compared with the single representative algorithm (*Repre*) and *WSNDiag*.

B. Literature review

Identification of the fault status (i.e., faulty or fault-free) of each processor (i.e. node) in a system based on a syndrome, the set of all test results, is termed system level diagnosis [46][47][48][49][50]. System level diagnosis was first introduced by Preparata, Metze, and Chien (*PMC model*) [51] to diagnose nodes in multi-processor computer

systems [52]. System level diagnosis in multi-processor systems became very useful for maintaining the reliability of the system that could otherwise be diminished by the increasing number of nodes and complexity of the system [53][54]. The theory of system level diagnosis was not limited to multi-processor systems but extended to various network environments: wired and wireless communication networks, and currently to wireless sensor networks.

Unlike fault-tolerant systems that use redundant modules (e.g., N-modular redundancy) or error detecting codes, to detect (or mask) faulty nodes, a testing method is used in system level diagnosis by assuming that each node can test other node(s). Each tester node tests its tested node(s) in a given system and the fault status of each tested node is determined based on the syndrome generated by an assumed fault model.

In [50], only a central observer made the diagnosis decision after analyzing the syndrome gathered from all tester nodes and distributed the system status (diagnosis information) to each fault-free node. This approach was not scalable with increasing network size.

With the advent of distributed diagnosis approaches (e.g., *Self* [55], *New self* [56], *Event_self* [57], and *Adaptive DSD* [58]), a reliable central observer that has global diagnosis information is no longer used [58]. Instead, each fault-free node can determine the fault status of other nodes in the system by direct testing and by exchanging the diagnosis information of other nodes with other fault-free neighbors.

Before the first adaptive algorithm [59] introduced by Hakimi and Nakajima, the testing assignment for a system was fixed and was not changed during execution

depending on the fault situation [60]. The adaptive algorithm [59] had a central observer like the *PMC model*. In the adaptive pattern, the testing assignment was changed depending on the results of previous tests [49]. Bianchini and Busken introduced an adaptive and distributed diagnosis algorithm called *Adaptive DSD* [58]. In the *adaptive DSD*, the testing topologies were changed depending on the fault situation. There was no limitation on the number of faulty nodes that could be diagnosed as long as at least one fault-free node existed. In *adaptive DSD*, each node was tested by only one fault-free node by using sequential testing. The diagnostic information was only disseminated in the reverse direction of tests performed. In the *Self* family of algorithms [55][56][57], each node is tested by at least $t+1$ other nodes (t is the number of faulty nodes). Since every fault-free node sends all test results of the tester nodes to all other fault-free nodes in the *New_self* algorithm [56], a large number of diagnostic messages must be forwarded through the network. By using *adaptive DSD*, the heavy message overhead of the *Self* family algorithms is reduced. Initially, each node only knows the test results of its tested node(s) but each node can know the fault status of all nodes after spending the necessary number of testing rounds for diagnosing the nodes in a system [58].

In 1995, Rangarajan, Dahbura, and Ziegler [9] introduced a new adaptive distributed diagnosis algorithm (RDZ) in an arbitrary network topology. The is one of the few diagnosis algorithms [9][46][61] for arbitrary network topologies in wired environments. The *adaptive DSD* and *Self* family algorithms previously introduced cannot be used for arbitrary network topologies, since they assume fully connected network topologies. Through the validation transaction [9] in the RDZ algorithm, nodes can be tested directly

by their tester, and also indirectly whenever the diagnosis information is disseminated. When a fault-free node receives a message, it must give an acknowledgement for that message within the specified time. If not, the sender learns that the receiver node failed. Thus, both static and dynamic faults can be detected. Faulty nodes can rejoin the network after being repaired at any time during algorithm execution [9].

Unlike *adaptive DSD* [58] where sequential testing was performed, testing in RDZ was performed by tester nodes without having any sequence restriction among the testers. Also, the fault detection and diagnosis message dissemination steps were separated and the dissemination steps were executed in parallel by flooding [9].

Since RDZ was introduced for point-to-point communication networks in wired environments, reducing information latency with increasing message overhead could be tolerable due to large bandwidth and power available. But this is not true in wireless sensor networks, since high message overheads would cause high energy consumption, and communication consumes most of the energy [12].

There are two published approaches [4][16] to trace faulty nodes in a wireless sensor network. One [16] is a centralized approach that maintains the global status only in a powerful base station and the other [4] is a distributed approach that maintains the fault status of all nodes in each node.

Staddon, Balfanz, and Durfee [16] introduced an algorithm that can trace the faulty nodes by using a powerful base station that has a global view of the network. The base station can build the routing topology of the entire sensor network by using the neighboring information received from each node. For this, each node recognizes its

neighbors when the initial route discovery protocol is run, and sends this neighbor information to the base station via neighbor nodes in order to save energy. Each node does not attempt to make a new route to the base station when it is needed, but receives the routing information from the base station directly. The base station is assumed to have enough transmitting power to send messages directly to all nodes [16]. Although each node can save energy by not directly attempting to make a new route, there is no way for a node to find a new route when the base station cannot give new routes to the nodes.

To trace the faulty nodes, the base station divides all nodes in the network into three groups: alive (i.e., fault-free), dead (i.e., faulty), and silent. Since the routing topology is a tree having the base station as the root, when a node in the middle of the path becomes faulty, the base station cannot receive any information transmitted from any of the nodes located below the faulty node. These nodes are called silent nodes since the base station cannot determine whether they are faulty or fault-free without more information.

By updating the routes near the known faulty nodes, the base station starts to determine which silent nodes are alive and dead. If all silent nodes are alive, the base station does not have to do route updating again since the nodes would respond to the base station by using the new routes obtained directly from the base station. But the route updating by the base station must be done repeatedly until all faulty nodes among the silent nodes are determined, since information from fault-free nodes cannot be delivered to the base station when some faulty nodes are intermediate nodes on the new paths. Thus, when there are a number of faulty nodes among the silent nodes, many

route updating procedures, message traffic, energy consumption and latency is required to deliver all sensing information from each silent but alive node.

Chessa and Santi [4] suggested a distributed crash fault diagnosis protocol (*WSNDiag*) designed for wireless sensor networks. Unlike fault diagnosis algorithms for the wired network environment that use one-to-one based testing, *WSNDiag* uses one-to-many based testing. Each fault-free node broadcasts an existence message, *IMA* (i.e., "I am alive" in [4]), to its neighbors. This message originates from an initiator when the diagnosis for the network is needed, and advertises each node's fault-free status to its neighbors. When a node does not receive an *IMA* message from a neighbor within the required time after broadcasting its *IMA* message, it considers the neighbor to be a faulty node. Diagnosis information from each fault-free node is aggregated and delivered to the initiator, the root of the spanning tree. Then the completed diagnosis information made by the initiator is disseminated to all nodes in the network.

A one-to-many broadcasting testing mechanism cannot provide reliable testing results. For better test results, one-to-one testing mechanism is preferred. Since *WSNDiag* assumes only static faults (i.e., no new faults occur during the execution of the diagnosis algorithm), faulty nodes that occur during the current algorithm execution cannot be diagnosed until the next algorithm execution. To deliver diagnosis information without having increased latency, an extra mechanism that handles dynamic failure events is required for *WSNDiag*. Since a tree is not maintained but is made per each use in *WSNDiag*, it consumes redundant energy without providing scalability.

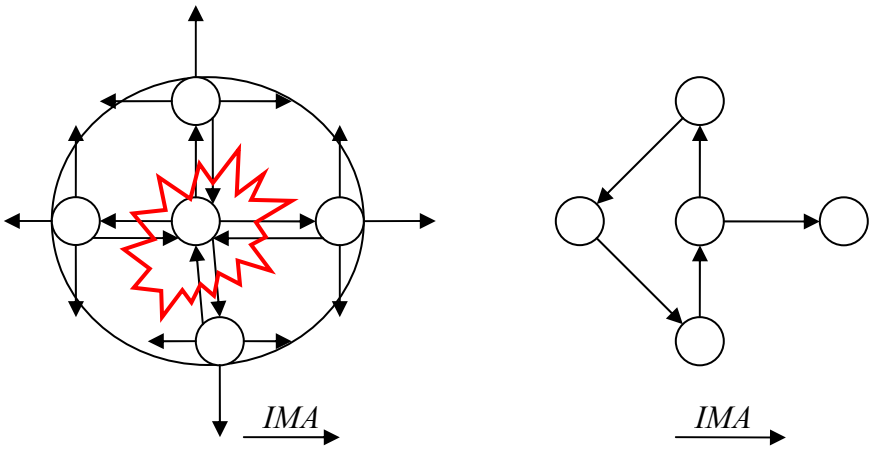
C. A new crash fault diagnosis algorithm for wireless sensor networks (Repre)

Each sensor knows its neighboring information when a routing tree is made after it is deployed in an environment. Based on this neighboring information, the initial tester-tested relationships among all nodes are established so that each tester node tests its tested node(s) regularly. And those relationships are adaptively changed depending on the faulty status of each node. After each regular testing, testers report the faulty nodes' information to the root node through the routing paths when they detect faults. At that time, dynamic faults in the network also can be detected and reported by using *INP* reconfiguration procedures locally maintaining the tree structure.

When a faulty node becomes fault-free (e.g., after battery recharge or repair) or a new node is deployed, it can join the network using the *JOIN* procedure introduced in section III, and find a tester using the *TESTME* mechanism described below.

1) Testing

While approaches like *WSNDiag* [4] use one-to-many testing that exploits the shared nature of wireless communication, one-to-one testing via unicasting mechanism is used in *Repre*. One-to-many broadcast testing used in *WSNDiag* [4] causes a high volume of incoming *IMA* messages from all fault-free neighbors and the receiving channel becomes a bottleneck (i.e., response explosion problem [62]). Fig. 69(a) shows the response explosion problem by using one-to-many testing and Fig. 69(b) shows one-to-one testing.



(a) One-to-many (b) One-to-one

Fig. 69. Different testing mechanism.

The hidden terminal problem [63] (resulting from collisions) can also occur since the RTS/CTS mechanism in a MAC protocol is not used while broadcasting. This kind of unreliable communication may cause incorrect testing results. Also in one-to-many testing, all neighbor nodes of a faulty node redundantly detect its faultiness and report it redundantly.

To increase the reliability of testing results and to reduce the redundancy, *Repre* uses unicasting for one to one testing. When the RTS/CTS/DATA/ACK mechanism [39] is used for unicasting in a MAC protocol [39][43][63], hidden terminal problems are avoided and collisions are reduced [63].

To make initial tester-tested relationships among nodes in the network, each tested node locally decides (randomly chooses) a neighbor as its tester node by unicasting a *TESTME* message to the node after getting all neighbors' information. When the

neighbor receives *TESTME*, it sends a *ACKTM* message back to confirm that it can be the tester.

After the initial relationships are made, each tested node unicasts an *IMA* message that tells its tester that it is fault-free in each testing round. This message becomes the basic testing method since the tested node uses multiple retransmissions in a MAC protocol when a MAC acknowledgement (ACK) of DATA (*IMA* message) is not received (DATA/ACK mechanism) within the timeout delay. In environments with a low packet loss rate, where retry in a MAC layer usually succeeds, this *IMA* message is enough for both tester and tested nodes.

If the tester node does not get an *IMA* message within the expected time, the tester assumes (considers) that the tested node is faulty and unicasts a *TEST* message to its tested node to confirm the faulty status. The tester node finally determines the faulty status of its tested node when the tester does not get the *REPLY* response message within the required time. If the tested node does not get a MAC acknowledgement *(ACK)* from its tester in response to an *IMA* message, it considers that the tester node is faulty and finds a new parent by broadcasting a *TESTME* message to its neighbors and waits for *ACKTM* responses. The node chooses its new tester by unicasting a *ACPTM* message to one of the responding nodes. If the previous tester is still alive and it receives a *TESTME* message from its tested node, it will also send an *ACKTM* to the tested node. In this case, the tested node continues using the previous tester node. By doing these, the tester (and tested) node doubly checks its tested (and tester) node so that the reliability of the test results is increased.

2) IAD message

The *Repre* algorithm also has a mechanism (*IAD*) where a tester node gives a precaution of its imminent death (e.g. battery depletion) to the neighbors. With this mechanism, the tested node(s) can find a new tester node without waiting for the next regular testing round. If a tested node sends an *IAD* message, its tester reports this information to the representative node by piggybacking on the data to be delivered. Since the tester is no longer expecting *IMA* message from the tested node and does not try to confirm the tested node's status, it saves energy and reduces the latency to recognize the fault.

If a child receives an *IAD* message from its parent, it would not attempt to send data to the current faulty parent, but instead would initiate dynamic reconfiguration procedures, saving energy and reducing message latency.

Fig. 70 shows the before and after situations when a node *X* that will soon die broadcasts an *IAD* message to its neighbors. When the tested node *Z* of node *X* receives the *IAD* message, it finds its new tester node *S* using the *TESTME* mechanism. When the tester node *Y* of node *X* receives the *IAD* message, it reports node *X*'s failure to the representative node before the next regular testing.

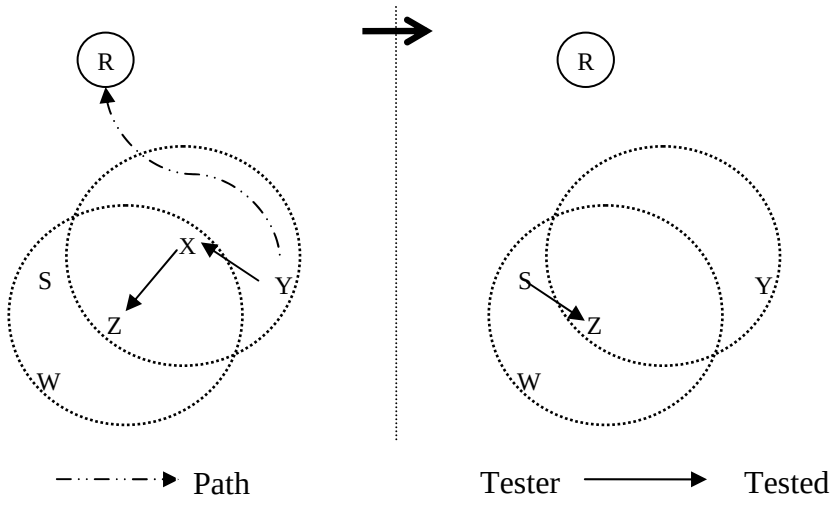


Fig. 70. Before and after situations when X sends *IAD* (*R* is root).

3) Top-to-bottom information dissemination

Nodes in a sensor network usually do not need to know the faulty status of all other nodes in the network. This is because the nodes in wireless sensor networks usually work in cooperation their neighbor nodes. However, there are some applications where each node needs to know the global status of the network. For example, in *WSNDiag* [4], when a mobile control observer wants to have a global view of network status from any fault-free node. For these situations, the diagnosis information that is delivered to the representative node is propagated to each node in the network through the paths in the tree.

4) Computational analysis

The proposed diagnosis algorithm *(Repre)* was analyzed for energy consumption and message overhead. Variables in Table I were used with additional variables in Table IV

for analysis. The analysis for establishing the initial tree is given in section IV.

TABLE IV
VARIABLES FOR COMPUTATIONAL ANALYSIS OF *REPRE, LOCAL,* AND *WSNDIAG*

Variable	Description
a	average number of neighbors
r	average number of children
L	total number of leaf nodes
L_event	total number of leaf nodes that have an event to be reported
event	total number of failure events
Testers	set of all tester nodes
Tested[j]	number of nodes tested from tester node *j*
FF_Tested[j]	number of fault-free nodes tested from tester node *j*
F_Tested[j]	number of fault nodes tested from tester node *j*
Chd_msg[j]	number of children that have a new event or an event needed to be delivered to its parent node *j*
Err[j]	number of communication error at node j
Err	total number of errors = $\|Tester\| \cdot Err[Tester]$

a) Testing

For testing, each tester *j* uses $(FF\_Tested[j]\text{-}Err[j])\cdot E(IMA_r)$ + $(F\_Tested[j]+Err[j])\cdot E(test_s)$ + $Err[j]\cdot E(reply_r)$ energy. Each fault-free tested node *k* uses $E(IMA_s)$ energy since each node is tested by only one fault-free node that requires a *IMA* message from its tested node. Each fault-free tested node whose *IMA* message was not delivered to its tester node would receive the *test* message from its tester and send the *reply* message as a response. It costs $E(test_r) + E(reply_s)$. Thus the total energy consumption for testing all nodes is expressed as (33).

$$\sum_{j=1}^{N} ((FF\_Tested[j]\text{-}Err[j])\cdot E(IMA_r) + (F\_Tested[j]+Err[j])\cdot E(test_s)$$

$$+Err[j]\cdot E(reply_r)) + |FF|\cdot E(IMA_s) + \sum_{j=1}^{N} Err[j] \cdot (E(test_r) + E(reply_s))$$

$$(\text{where, } j \in FF \cap Testers) \quad (33)$$

It can also be formulated as (34) in the worst case when all fault-free nodes are testers. Each node tests exactly one node and each node is tested by one node in this case.

$$|FF|\cdot E(IMA_s) + (|FF|-Err)\cdot E(IMA_r)) +$$

$$(|F|+Err)\cdot E(test_s) + Err\cdot (E(test_r) + E(reply_s) + E(reply_r)) \quad (34)$$

The total number of messages used is $|FF| + |F| + 2Err$ since there are $|FF|$ *IMA* messages, $|F|$ + Err *test* and Err *reply* messages.

b) Information dissemination

The analysis is based on the policy that whenever diagnosis information (*info*) is delivered to another node, the node that receives the information must give an acknowledge message *(ackinfo)* back to the sender. So the sender makes sure that the message is successfully delivered to the receiver. Although this application level acknowledgement consumes more energy than the MAC acknowledgement, it can be used for a network that needs higher delivery confirmation. In a network that does not need an *ackinfo* message back to the sender, $E(ackinfo_{s+r})$ should be omitted from the following analysis.

By default, information transmissions from nodes to the representative (bottom-to-top) occur and the energy cost for those transmissions is analyzed. For disseminating diagnosis information, three different locations (i.e., leaf, internal, and representative) of each node must be considered. When a node is a leaf of the tree and it has an event to

report to its parent, the node consumes $E(info_s) + E(ackinfo_r)$. Thus energy consumption of all leaf nodes that have an event to report is expressed as (35).

$$L\_event \cdot (E(info_s) + E(ackinfo_r)) \quad (35)$$

It also can be formulated as (36) in the worst case when all faulty nodes are detected by leaf nodes.

$$event \cdot (E(info_s) + E(ackinfo_r)) \quad (36)$$

When a node j is an internal node of the tree and it receives messages from some children and sends them to its parent, it spends the energy given in (37) since node j aggregates event information obtained from its children and sends the aggregated result to its parent.

$$chd\_msg[j] \cdot (E(info_r) + E(ackinfo_s)) + E(info_s) + E(ackinfo_r) \quad (37)$$

In the worst case, each internal node j sends *info* to its parent each time when it receives an event from any child without aggregating with other event information. Thus, the total amount of energy used by all internal nodes for this worst case is expressed as (38).

$\sum_{i=1}^{event} (h[L_i]-1) \cdot (E(info_r) + E(ackinfo_s) + E(info_s) + E(ackinfo_r))$ (at here, $h[L_i]$ is hop counts between a leaf L_i that reports $event_i$ and the representative R) (38)

The representative node R consumes (39) when it receives information from its children that have an event to relay or event detected:

$$chd\_msg[R] \cdot (E(info_r) + E(ackinfo_s)) \quad (39)$$

In the worst case, when the node R receives nonaggregated *info* of each event from

any child, the total energy used in node R is expressed as (40).

$$event \cdot (E(info_r) + E(ackinfo_s)) \quad (40)$$

Thus, the total worst case energy consumption for reporting failure events from detecting nodes to the representative node R is (36) + (38) + (40).

In the worst case, the total number of messages used is $O(event \cdot h)$ (h is the height of the tree).

c) Top-to-bottom transmission

Optionally, when each node needs to receive diagnosis information from the representative node R, node R broadcasts the diagnosis information to its children. And the children broadcast this information to their children, and so on until the diagnosis information reaches the leaf nodes.

In this case, the representative node consumes $E(info_s) + chd[R] \cdot E(ackinfo_r)$. Each internal node j consumes $E(info_r) + E(ackinfo_s) + E(info_s) + chd[j] \cdot E(ackinfo_r)$. Each leaf node consumes $E(info_r) + E(ackinfo_s)$. Thus the total energy consumption for all nodes to receive the diagnosis information is expressed as (41):

$$E(info_s) + chd[R] \cdot E(ackinfo_r) + L \cdot (E(info_r) + E(ackinfo_s)) + (N-L-1)(E(info_r) + E(ackinfo_s) + E(info_s) + chd[j] \cdot E(ackinfo_r)) \quad (41)$$

The message overhead used for delivering diagnosis information to every node from the representative R is $O(N)$.

d) Children information dissemination

This subsection describes the energy cost for the representative node to receive

children's information from all nodes. It is used when the representative node wants to know the information of all fault free nodes.

Depending on the location of each node in the tree, a different amount of energy is used. Each leaf node consumes $E(childinfo_s) + E(ackchildinfo_r)$ energy. Since leaf nodes do not have children, they send a *childinfo* message that only contains "-1" in the *N_ch* field (Appendix A) of the message. Since each intermediate node *j* receives and aggregates the information from all children and sends it to its parent, it consumes $chd[j] \cdot (E(childinfo_r) + E(ackchildinfo_s)) + E(childinfo_s) + E(ackchildinfo_r)$ energy. Since the representative node only receives information from its children, it consumes $chd[R] \cdot (E(childinfo_r) + E(ackchildinfo_s))$. Thus, the total energy consumption for delivering all children's information to the representative node is expressed as (42):

$$\begin{aligned} & L \cdot (E(childinfo_s) + E(ackchildinfo_r)) \\ & + (N-L-1) \cdot (chd[j] \cdot (E(childinfo_r) + E(ackchildinfo_s)) + E(childinfo_s) + \\ & E(ackchildinfo_r)) \\ & + chd[R] \cdot (E(childinfo_r) + E(ackchildinfo_s)) \end{aligned} \quad (42)$$

The message overhead used for delivering children's information to the representative *R* is $O(N)$.

$E(childinfo_s)$, and $E(childinfo_r)$ have different values depending on the amount of information. When the children's information is delivered up to the representative node, each intermediate node aggregates all children's information, and delivers the aggregated information to its parent node. Thus, nodes higher in the tree deliver more information. The representative node receives all children's information.

In worst case analysis with in an *r*-ary complete tree [64] that generates the maximum packet size on each level in the tree, the total packet size generated by all leaf nodes is r^h. The height of the tree $h = log_r(N(r-1)+1)/r)$, since each leaf node sends a *childinfo* packet with a single integer as payload.

Each internal node at depth *h-i* (*i* is from 0 to *h*) generates a *childinfo* message whose size is $((r+1)(r^i-1))/(r-1)$. The number of nodes at depth *h-i* is r^{h-i}. Thus the total packet size generated for delivering all children's information to the representative is expressed as (43). The average packet size is $(43)/(N-1)$.

$$r^h \cdot (2B+9B) + \sum_{i=1}^{h-1} r^{h-i} \cdot (2B \cdot (r+1)(r^i-1))/(r-1)+9B) \quad (43)$$

(where, $9B=|AppHdr+CRC|$ for the *childinfo* packet format in Appendix A)

5) Competitive analysis

The proposed diagnosis algorithm (*Repre*) is compared with *WSNDiag* [4]. The same parameters except number of nodes that were used in section IV were used in this analysis.

Fig. 71 shows the accumulated energy consumption of each method by increasing number of executions. At the first execution, *Repre* is a little bit more expensive than *WSNDiag* for both network sizes, because it consumes energy for testing, local configuration, and fault reporting after making the initial tree. On the other hand, *Repre* consumes less energy than *WSNDiag* after two executions. This is because *WSNDiag* completely rebuilds the communication tree each time, while *Repre* only reconfigures the existing tree as needed. Thus, the gap between *Repre* and *WSNDiag* becomes larger

as the network size increases.

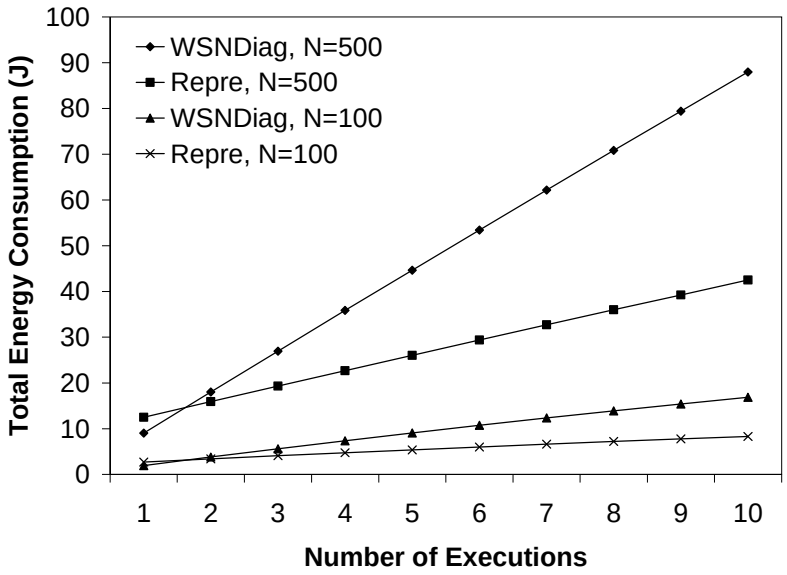


Fig. 71. Energy consumption (a=10, r=3, 3 faults/exec.).

Fig. 72 shows the non-accumulated energy of each method consumed at each execution. In *WSNDiag*, there is little different between one execution and another. In *Repre*, there is a big difference between the first execution and later ones. This is because the tree is made only at first execution in *Repre*. When the tree is increased to 500 nodes, the energy gap consumed becomes much bigger in *WSNDiag* rather than in *Repre*. Since the number of remaining fault free nodes is continuously decreased by three, the amount of energy at each execution is also decreased.

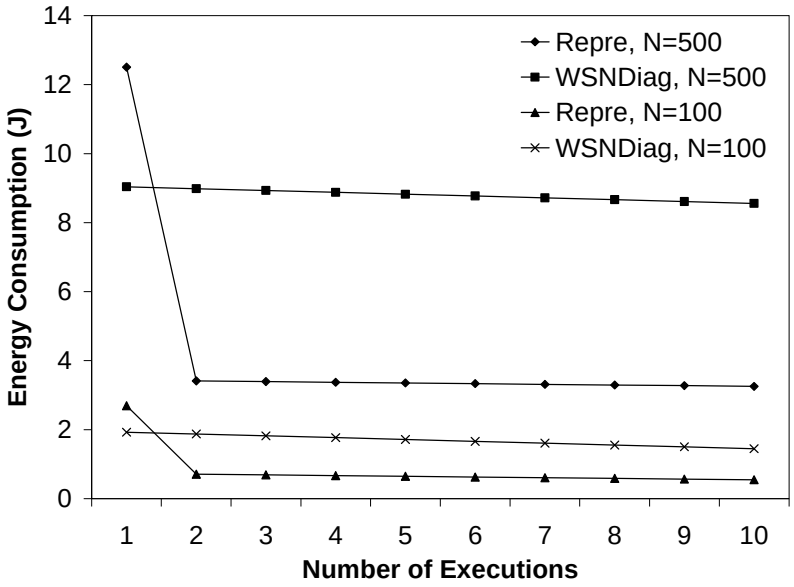


Fig. 72. Energy consumption (non-accumulated, a=10, r=3, 3 faults/exec.).

Fig. 73 shows the total energy consumed for each different operation when the first execution is done in *Repre*. When the network size is increased, the energy consumed for making a tree increases rapidly since all nodes consume energy for that. In contrast to making a tree, the other operations consume much less energy. Since the energy is needed only locally for a new path to the parent, the energy for local configuration does not depend on the total number of nodes in the network, but on the number of neighbors. Energy for fault reporting depends on the number of relaying nodes between a detecting node and the representative. Since each node has 3 children there is not much height difference network sizes 100 to 700. Thus, energy consumption increases only a small amount as the network size is increased.

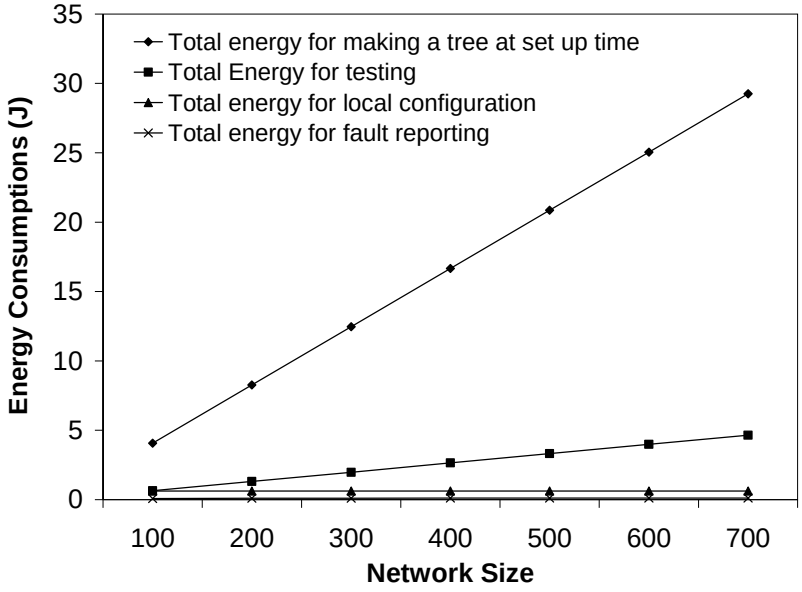


Fig. 73. Energy consumption for *Repre* (first exec., N=500, a=25, r=3, 3 faults/exec.).

Fig. 74 shows the energy for first execution and five executions for both approaches for different network sizes. At first execution, *WSNDiag* always consumes less energy than *Repre* since *Repre* needs more operations. But the energy gap between the first execution energy and five executions for *WSNDiag* is much larger than *Repre* since it makes a new tree each time.

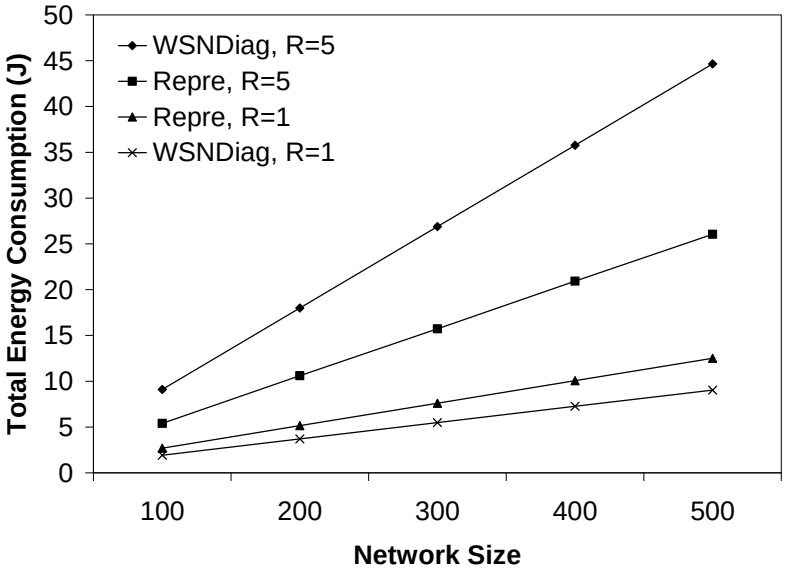


Fig. 74. Energy consumption (a=10, r=3, 3 faults/exec.).

From Fig. 75, we know that *WSNDiag* is more affected by the number of neighbors than *Repre*. This is because *WSNDiag* uses a one-to-many testing approach and makes a tree for each execution. For *Repre*, there is no such consumption since it uses one-to-one testing that is not much affected by the number of neighbors. Also by only building a tree at first execution, it uses much less energy.

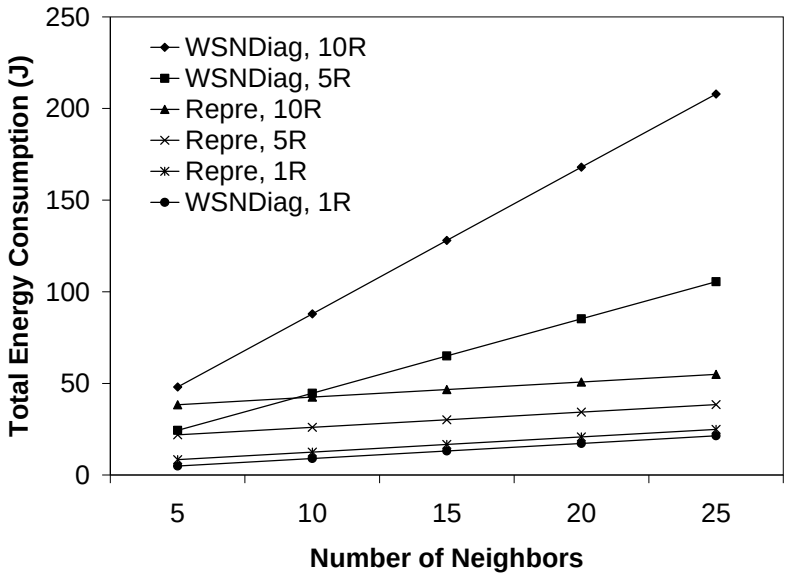


Fig. 75. Energy consumption (N=500, r=3, 3 faults/exec.).

Fig. 76 shows the energy consumption for different executions with varying number of faults from 1 to 4. The number of faults has little impact on the amount of energy consumed in both *WSNDiag* and *Repre* because most energy is consumed making the tree. But with varying number of faults, when the number of executions is increased, the total number of fault free nodes is decreased proportionally with the number of faults. For *WSNDiag*, the number of fault free nodes that consumed energy for making a tree is continuously decreased with increasing faulty nodes. In *WSNDiag*, after 10 executions are done, the consumed energy for 4 faults per execution is less than the energy used for 3 or fewer faults per execution. *Repre* is less sensitive to the number of faults since it does not rebuild the tree for each execution.

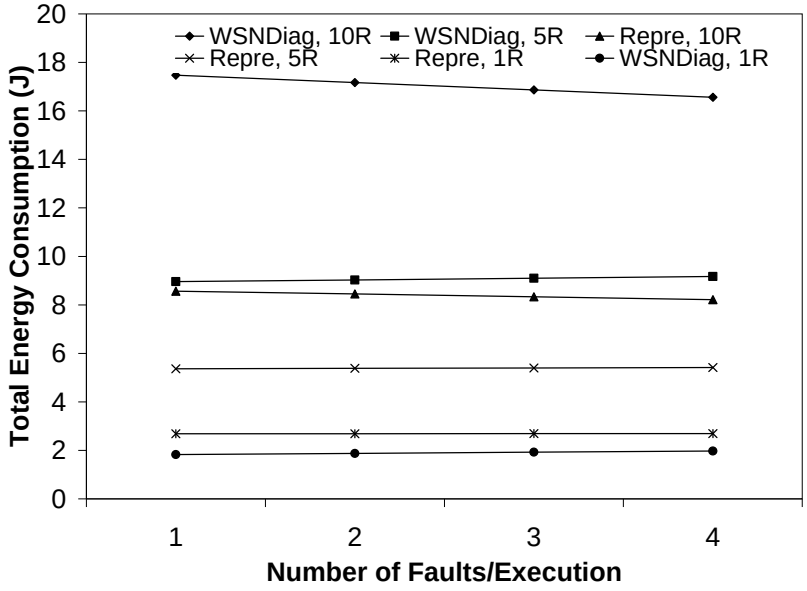


Fig. 76. Energy consumption (N=100, r=3).

Fig. 77 shows energy consumption per node in *Repre* with different numbers of neighbors for each different network size. Since the transmission power is assumed to be constant, the transmission range cannot be changed with increasing node density. Since the number of neighbors increases proportionately with increasing number of nodes, there is not much difference in per-node energy except the first execution. For the first execution, since all nodes broadcast a message to the neighbors to make an initial tree, both the number of nodes and the number of neighbors affects the energy cost. Thus it costs the highest when N=700 and a=35. Fig. 77 also includes energy consumption per node for only testing at each execution. Since all nodes are testing and only the nodes that are on or near the information dissemination path(s) are involved with local reconfiguration and data dissemination, almost all energy consumption per node is for

testing, except the first execution.

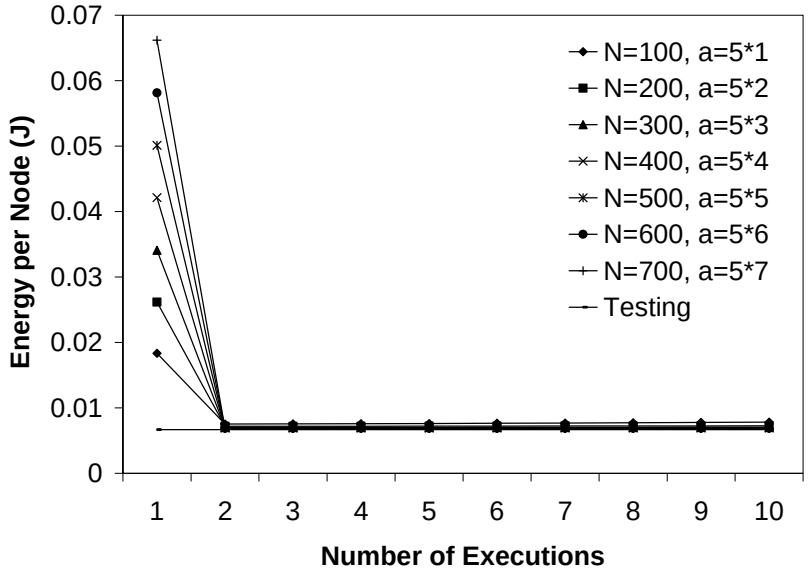


Fig. 77. Energy consumption per node (*Repre*, r=3, 3 faults/exec.).

Fig. 78 includes the *WSNDiag* approach with the conditions of Fig. 77. When *WSNDiag* is used, a tree is always made at each execution. Thus energy per node at each different condition is not much different among different executions. In *WSNDiag*, energy consumption is proportionally increased with increasing total number of nodes and neighbors since the number of neighbors increases proportionately with increasing number of nodes, as in Fig. 77.

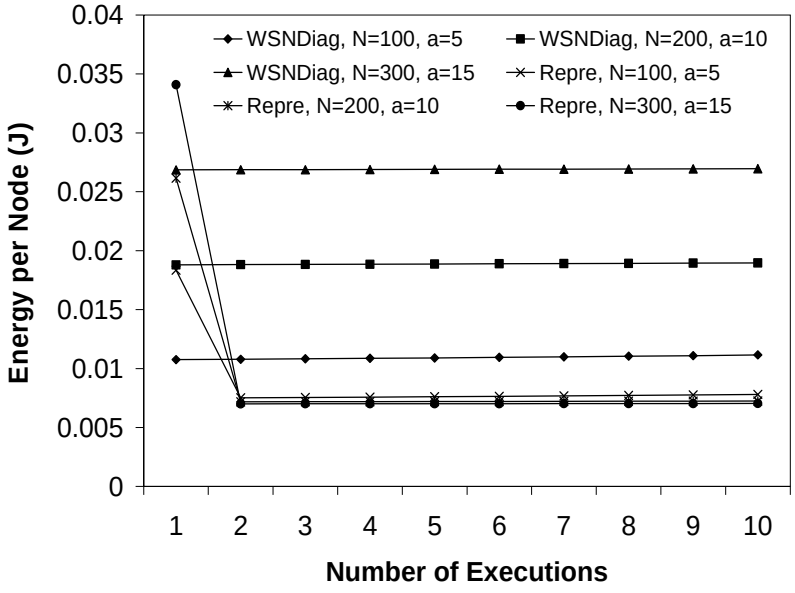


Fig. 78. Energy consumption per node (r=3, 3 faults/exec.).

In the *Repre* analysis, all fault free nodes are assumed tested by their testers, by exchanging *TEST* and *REPLY* messages. If only tester nodes that did not get *IMA* message(s) from its tested node(s) send a *TEST* message to each tested node and wait for *REPLY* message(s), much energy can be saved. Fig. 79 shows energy consumption when different percentages of fault free nodes are testing by exchanging *TEST* and *REPLY* messages.

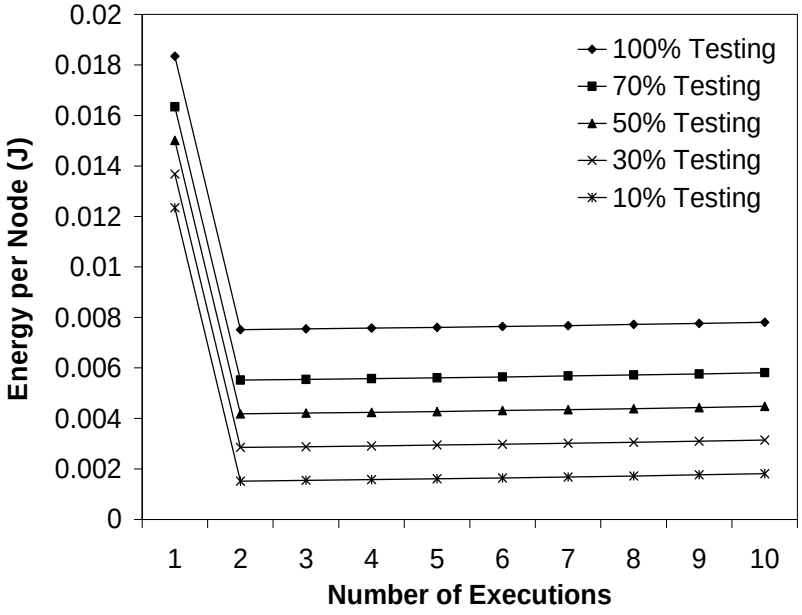


Fig. 79. Energy consumption for testing (N=100, a=5).

Fig. 80 and Fig. 81 show total energy consumption for the *Repre* and *WSNDiag* approaches respectively, with the same conditions as Fig. 77 and Fig. 78. When *WSNDiag* is used, the amount of energy grows much faster than *Repre* (notice the difference in the *y*-axis scales).

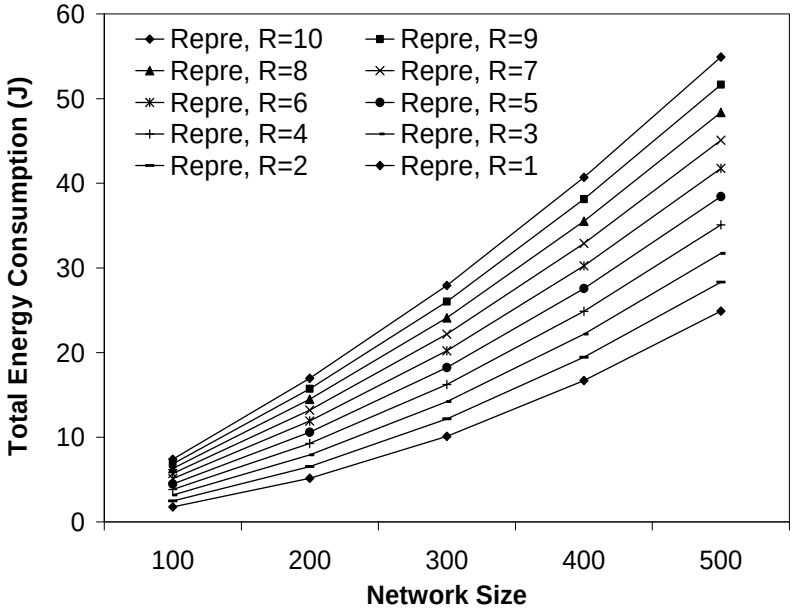


Fig. 80. Energy consumption for *Repre* (a=5*(size / 100), r=3, 3 faults/exec.).

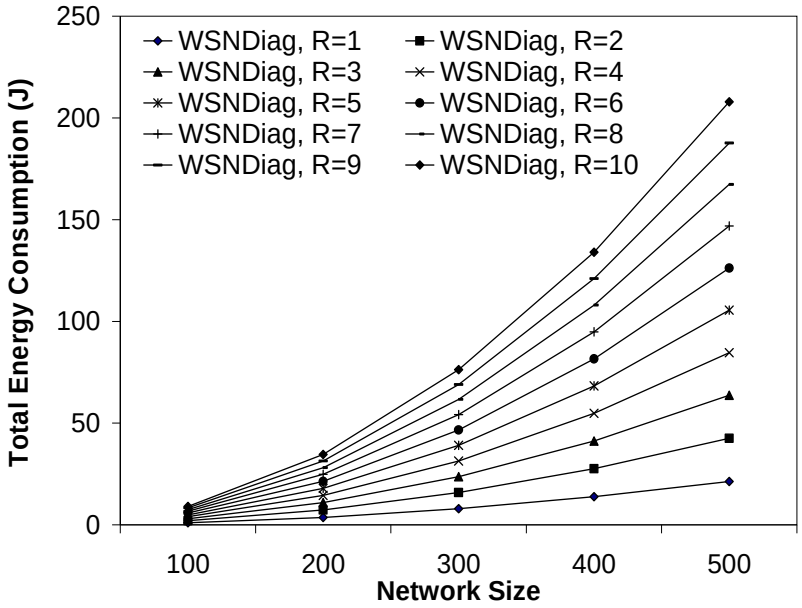


Fig. 81. Energy consumption for *WSNDiag* (a=5*(size / 100), r=3, 3 faults/exec.).

Fig. 82 shows how the fixed listening period *C* prior to testing or disseminating data affects on energy consumption for both approaches. In comparison to normal operations such as testing or disseminating, a large amount of energy is spent on listening. By reducing this period as much as possible, energy consumption can be significantly reduced.

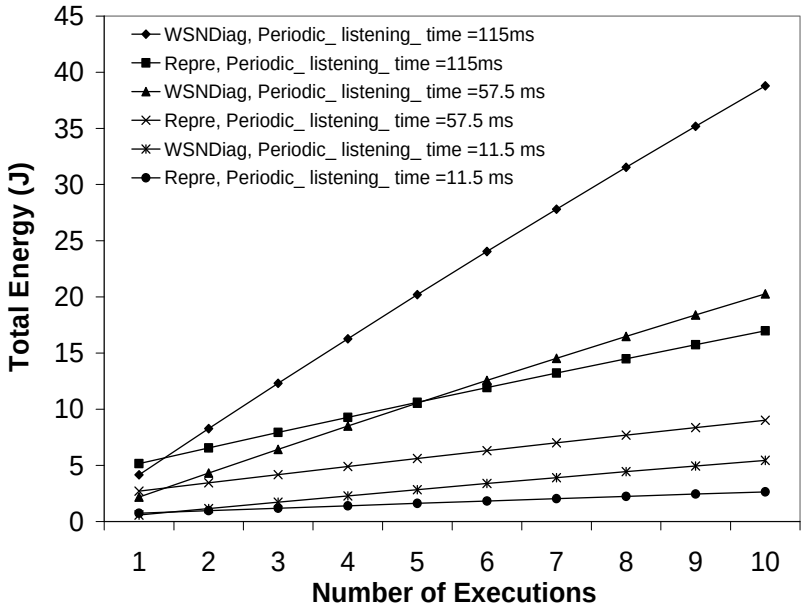


Fig. 82. Energy consumption (N=200, a= 10, r=3, 3 faults/exec.).

Fig. 83 shows the energy consumption for different node types in *Repre* for different numbers of executions. For leaf and internal nodes, the average value among all nodes of that type is shown. As discussed earlier, leaf nodes always consume less energy per algorithm execution. The internal nodes consume more energy than leaf nodes since leaf nodes only send information to the internal nodes but the internal nodes send the

diagnosis information to the representative after receiving it from leaf nodes. The representative node always consumes more than both leaf and internal nodes since all information ultimately arrives at the representative node. This suggests that periodically the tree should be rebuilt so that the internal and representative nodes become leaves and vice versa, in order to average out per-node energy consumption. In practice the representative node is likely to be a high-energy node, so the exchange should take place between leaves and internal nodes.

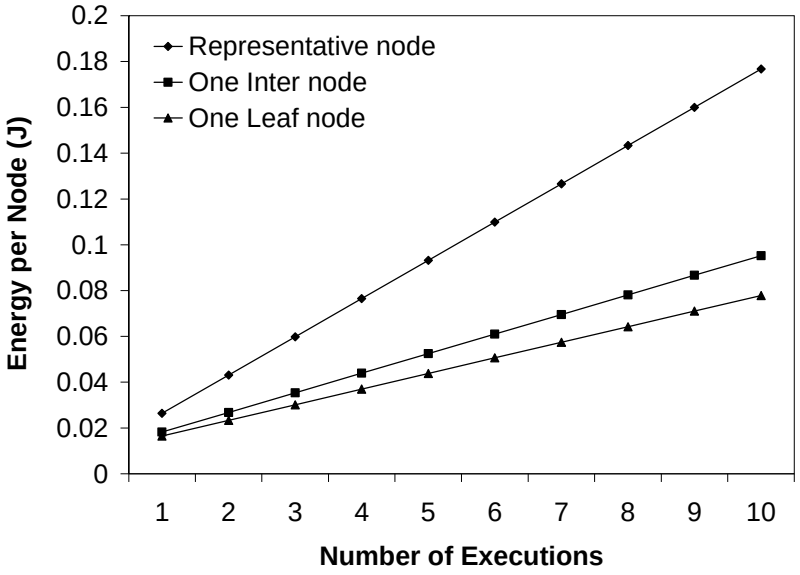


Fig. 83. Energy consumption per node (*Repre*, accumulated, N=200, a=10, r=3, 3 faults/exec.).

Fig. 84 shows energy consumption per node in *Repre* for different numbers of executions. Unlike Fig. 83, this shows the non-accumulated value for each execution. Since the conditions are the same at each execution with 10 Neighbors, 3 Children and 3 Faults, each node consumes almost the same energy for execution, except the first

execution that needs extra energy to make a tree.

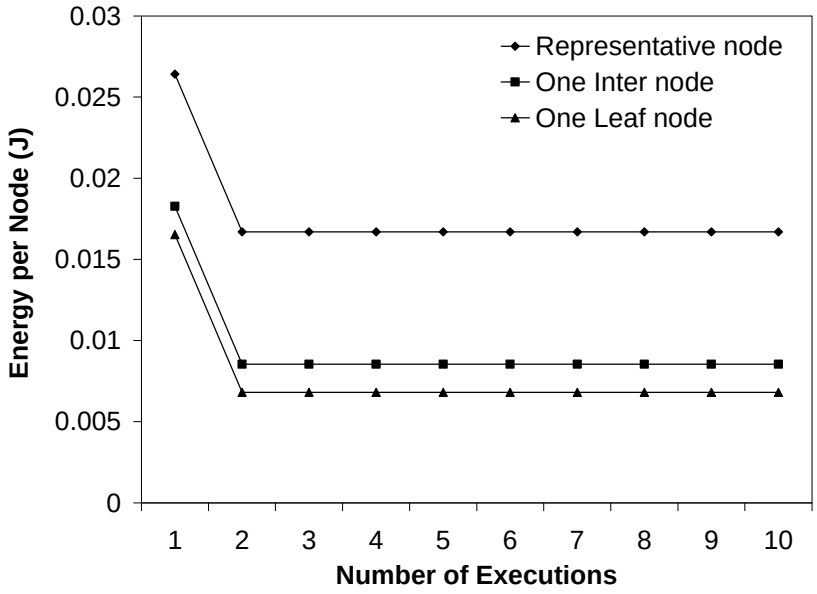


Fig. 84. Energy consumption per node (*Repre*, non-accumulated, N=200, a=10, r=3, 3 faults/exec.).

D. A scalable fault diagnosis algorithm (Local)

This subsection describes *Local*, an extension of the *Repre* approach to provide scalability in a large network. The network is divided into zones. Each zone has its own local tree and the root of the local tree becomes the local representative node of the local tree. Each root of the local tree is formed based on relative hop distance in the initial routing tree.

Fig. 85 shows a sensor network with 60 randomly located nodes. There are five zones in the sensor network, from 1 to 5. Each local root (H2, H3, H4, and H5) is chosen either when the initial tree is made or later. Initial representative node *R* becomes the highest local root H1. The diagnosis procedure is executed within each zone and the diagnosis

information is sent to the local root of the zone. These local roots become leaves of the next higher local tree and report their diagnosis summary to its local representative node, and so on until the initial representative node is reached.

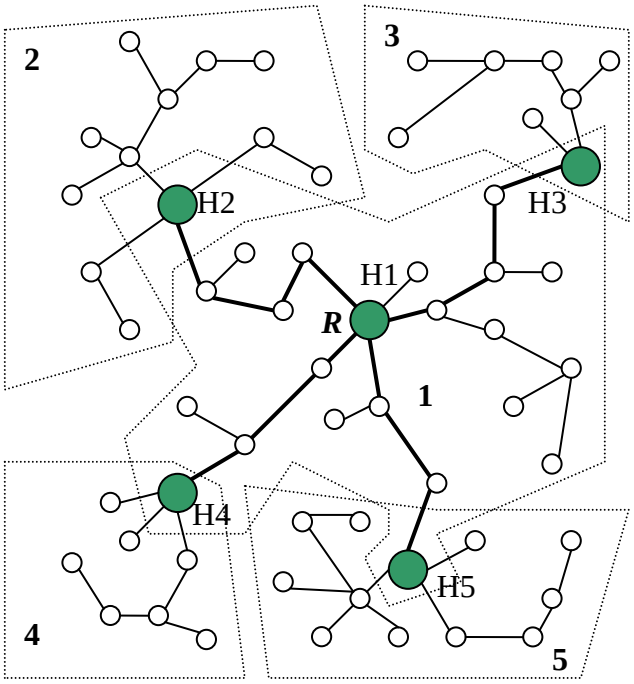


Fig. 85. Zone based sensor network.

1) Choosing the local representative nodes

It is assumed that the control observer knows the proper number of local roots in a given total number of nodes and the relative hop distance (*d*) to the next level of the local roots. In Fig. 85, H2, H3, H4, and H5 are the next lower level local roots of the initial root H1. The control observer can estimate these values through experience or pre-computation.

When a *PARENT* message is broadcast from the initial root node, distance (*d*)

information is included along with the hop count (hop count = 0) and a zone name (initial root ID). Since each node *j* can know its hop count when it receives a *PARENT* message from a neighbor, it declares itself a local representative if $h_j \ mod \ d = 0$ (where, h_j is the hop count from initial root at node *j*). It then includes its zone name (its ID) in the *PARENT* message. If a node is not a local root, then it has the same zone name as its parent's zone name. This process is repeatedly executed until all nodes in the network know their zones.

Even though each local root does not know how many nodes are in the zone at set up time, it can know that when it receives the first diagnosis results from all nodes in the zone. Unlike a tester node that only reports diagnosis information when it detects a faulty node, all nodes report their test results in the first testing round.

When the control observer estimates the *d* value after getting the initial tree information, leaf nodes send *NRH* messages (Appendix A) to their parents after making an initial tree. An *NRH* message describes the total number of descendants (*N*), average number of children for each parent (*R*), and the height of the node (*H*). Each node sends an *NRH* message that aggregates the *NRH* messages received from descendants with its local information to its parent. This process is repeatedly executed until the control observer receives *NRH* messages from all children.

When the *d* is distributed from the control observer to all nodes in the network, since each node *j* knows its maximum hop count from the leaf nodes (i.e., height), it declares itself a local root if $h_j \ mod \ d = 0$. In this way, each local root is located as many as *d* hops away from the leaf nodes, and so on up the tree, considering the local roots as

leaves if multiple levels of local representatives are needed. A zone based sensor network in Fig. 1 is made in this way with $d = 4$.

2) Testing

Tester-tested relationships are established and adaptively changed against faulty nodes among the nodes within the same local tree (zone) by using the *TESTME* mechanism. The mechanism is restricted to the zone is because the diagnosis information of the local tree is delivered to the local root through the paths between the testers and the local root in the local tree. For this, the local zone information is added to the control packets (i.e., *TESTME, ACKTM,* or *ACPTM*) that are used in the *TESTME* mechanism.

3) Information dissemination

When the diagnosis procedure is executed in a zone, each local root gathers diagnosis information from its zone and sends only the summarized information (e.g. number of faulty nodes in a zone) to the control observer *R* when it needs attention. The local root node sends the local diagnosis information to the descendants of the local tree when each node needs to know the diagnosis information for the nodes in the same zone.

Each local root communicates with other local roots through the paths (i.e., one local root to the initial root and the initial root to the other local root) when a node changes its current local tree to other local tree.

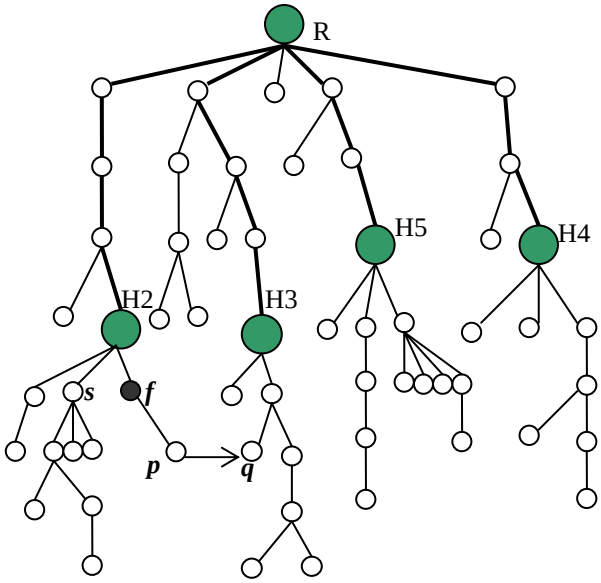


Fig. 86. Hierarchical shape of Fig. 85.

For example, in Fig. 86 that represents Fig. 85 in a hierarchical shape, a node *p* belonging to zone 2 finds its parent *f* faulty. When there is no neighbor that can become its new parent within zone 2, it considers a neighbor *q* in zone 3 as its parent. Node *p* changes its zone name to zone 3 and sends its previous zone name 2 and the death of node *f* to its new parent *q*. Then node *q* forwards this information to its local root H3. H3 in turn informs H2 about the reconfiguration through H3 to *R* and *R* to H2. H2 then deletes *p* and *f* from its list of descendants. The failure of node *f* can also be known to H2 from its tester.

4) Computational analysis

An analytical energy model that is used for *Local* is described below. Variables in Tables I and IV were used with additional variables in Table V for analysis.

TABLE V
ADDITIONAL VARIABLES FOR COMPUTATIONAL ANALYSIS OF *LOCAL*

Variable	Description
L	total number of leaf nodes in a local tree
l_event	total number of leaf nodes that have an event to be reported in a local tree
N	total number of nodes in a local tree
T	number of local trees
D	maximum hop distance from each node to its local root in a local tree
$depth_j$	depth of node j in initial tree (depth of initial representative R, $depth_R = 0$)

a) Set up local trees

When using the *NRH* message for setting local roots, there are two steps needed to select local representatives. The first step is delivering an *NRH* message to the initial representative. The initial representative consumes $chd[R] \cdot (E(acknrh_s) + E(nrh_r))$, each intermediate node j consumes $chd[j] \cdot (E(nrh_r) + E(acknrh_s)) + E(nrh_s) + E(acknrh_r)$, and each leaf node consumes $E(nrh_s) + E(acknrh_r)$. Thus, total energy consumption for delivering *NRH* message is expressed as (44).

$$chd[R] \cdot (E(acknrh_s) + E(nrh_r)) + L \cdot (E(nrh_s) + E(acknrh_r)) +$$
$$(N-L-1) \cdot (chd[j] \cdot (E(nrh_r) + E(acknrh_s)) + E(nrh_s) + E(acknrh_r)) \quad (44)$$

The second step is delivering a *HOP* message that has distance d value from the initial representative to all nodes. Energy consumption for delivering this message is expressed as (45).

$$E(hop_s) + chd[R] \cdot E(ackhop_r) \text{ (in representative } R)$$
$$+ (N-L-1) \cdot (E(hop_r) + E(ackhop_s) + E(hop_s) + chd[j] \cdot E(ackhop_r)) \text{ (in all intermediates)}$$

$$+ L\cdot(E(hop_r) + E(ackhop_s)) \text{ (in all leaves)} \quad (45)$$

The message overhead used for delivering the *NRH* message or *HOP* message is *O(N).*

After learning its local root from the *HOP* message, each node sends its children information to its parent and so on up to the local root. Energy consumption for that is expressed as (46).

$$l\cdot(E(childinfo_s) + E(ackchildinfo_r)) \text{ (in local leaves)}$$
$$+ (n\text{-}l\text{-}1)\cdot(chd[j]\cdot(E(childinfo_r) + E(ackchildinfo_s)) + E(childinfo_s) + E(ackchildinfo_r))$$
$$\text{(in all local intermediates)}$$
$$+ chd[R_{local}]\cdot(E(childinfo_r) + E(ackchildinfo_s)) \text{ (in a local root)} \quad (46)$$

Thus total energy consumption for all local representative nodes to receive their own children information is $T\cdot(46)$.

In worst case analysis with in an *r*-ary complete tree [64], (46) becomes (47). In an *r*-ary complete tree, $T=\lceil((r^d)^{h/d}-1)/(r-1))\rceil$.

$$r^d\cdot(E(childinfo_s) + E(ackchildinfo_r)) \text{ (in local leaves)}$$
$$+ ((r^{d+1}-1)/(r-1)-r^d-1)\cdot(r\cdot(E(childinfo_r)+E(ackchildinfo_s))+E(childinfo_s) +$$
$$E(ackchildinfo_r)) \text{ (in all local intermediates)}$$
$$+ r\cdot E(childinfo_r) + E(ackchildinfo_s) \text{ (in a local root)} \quad (47)$$

Thus for setting up the local trees, as much as (44) + (45) + $T\cdot$(46) more energy is consumed than the energy consumed for making a single representative tree shown in section IV.

The total packet size generated for delivering all children information to each representative in each zone is (48). The average packet size is $(48)/((r^{d+1}-1)/(r-1)-1)$.

$$r^d \cdot (2B+9B) + \sum_{i=1}^{d-1} r^{d-i} \cdot (2B \cdot (r+1)(r^i-1))/(r-1)+9B) \quad (48)$$

b) Testing

The analysis used in *Repre* (i.e., (33)) is applied to each zone. The total energy for all zones is T times the amount consumed for one zone. When each parent becomes a tester of its children in an r-ary complete tree [64], the number of local fault free tester and fault free tested nodes are maximally $\sum_{j=0}^{d-1} r^j$ and $\sum_{j=1}^{d} r^j$.

c) Information dissemination

By default, bottom-to-top information transmission to the initial representative node R is used and the energy cost for that is analyzed. Total energy consumption for a fault detecting node to send diagnosis information to its local root is expressed as (49).

$$E(info_s) + E(ackinfo_r) \text{ (in a detecting node)}$$
$$+(d-1)\cdot(E(info_r) + E(ackinfo_s) + E(info_s) + E(ackinfo_r)) \text{ (in internal nodes)}$$
$$+(E(info_r) + E(ackinfo_s) \text{ (in local root)} \quad (49)$$

In the worst case, when diagnosis information is delivered without aggregation with other diagnosis information, total energy used in all local trees to deliver the diagnosis information is *event · (49).*

The total energy consumption for delivering the summarized information from a local root j to the representative R is expressed as (50).

$$(depth_j - 1)\cdot(E(info_r) + E(ackinfo_s) + E(info_s) + E(ackinfo_r)) \text{ (in internal nodes)}$$
$$+ E(info_s) + E(ackinfo_r) \text{ (in a local root j)}$$

$$+ E(info_r) + E(ackinfo_s) \text{ (in representative R)} \quad (50)$$

The total energy consumption for delivering information from the representative *R* to a local root *j* is expressed as (51).

$$(depth_j - 1)\cdot(E(info_r) + E(ackinfo_s) + E(info_s) + E(ackinfo_r)) \text{ (in internal nodes)}$$
$$+ E(info_s) + E(ackinfo_r) \text{ (in representative j)}$$
$$+ E(info_r) + E(ackinfo_s) \text{ (in a local root j)} \quad (51)$$

When the main representative node *R* sends information to all local representatives, the energy consumption when using an *r*-ary complete tree [64] for the worst case analysis is expressed as (52).

$$E(info_s) + r\cdot E(ackinfo_r) \text{ (in node R)}$$
$$+ r\cdot(r^{h-d}-1)/(r-1)\cdot(E(info_r) + E(ackinfo_s) + E(info_s) + r\cdot E(ackinfo_r)) \text{ (in all internal nodes between R and the lowest local roots)}$$
$$+ (r^{h-d}-1))\cdot(E(info_r) + E(ackinfo_s)) \text{ (in the lowest local roots)} \quad (52)$$

5) Competitive analysis

The proposed diagnosis algorithm (*Local*) is compared with the single Representative algorithm (*Repre*) and *WSNDiag* [4]. In this analysis, in each testing execution, each tester node is assumed to send a *TEST* message to its tested node without waiting for an *IMA* message from its tested node and the fault free tested node is assumed to send a *REPLY* response message back to its tester node. In *Repre*, tester and tested relationships are made among any neighbors while each parent becomes the tester of all its children in *Local*. In *Repre*, all *TEST* and *REPLY* messages are unicasted from each tester and the fault free tested nodes. In *Local*, each tester node broadcasts the *TEST* message to its

children and each fault free child unicasts a *REPLY* message to the parent, the tester. This is reliable, since if a child did not receive the *TEST* broadcast and respond with a *REPLY*, the parent can retry. The parameters in Table VI were used in the analysis. Transmit and receive powers were taken from [12].

TABLE VI
PARAMETERS FOR COMPUTATIONAL ANALYSIS OF *REPRE, LOCAL,* AND *WSNDIAG*

Parameter	Description
Transmit power	0.6W
Receive power	0.2W (33% of transmit power)
Bandwidth	2 Mbps
MAC	S-MAC
Number of nodes	121, 364, 1093, 3280, 9841, 29524

Fig. 87 shows the energy consumption for different numbers of zones in *Local*. There are 29524 nodes and each node has 3 children. This initial tree is a 3-ary complete tree with height 9. The energy consumption for the first execution is the same for all hop counts (*d)* since the same energy was consumed for deciding *d* and delivering children information to each local representative node. For this network, minimum energy is consumed when *d* is 3, 4, or 5. This is true for any number of cumulative executions. When *d* is 9, there is only one tree, which consumes more than twice the energy of an optimal solution.

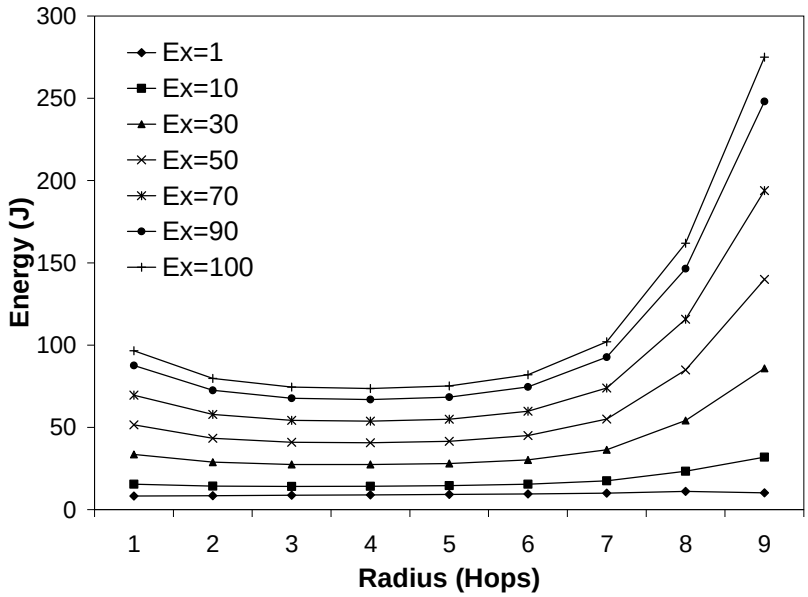


Fig. 87. Cumulative energy consumption in *Local* (N=29524, a=10, r=3, 3 faults/exec.).

Fig. 88 shows the cumulative energy consumption for different number of executions from 1 to 100. *Local* with different *d* and *Repre* were compared. *Repre* consumed more energy than *Local* for all *d*. Although Local with *d* = 9 has a single representative like *Repre*, it consumes less energy than *Repre*. This is because each tester and tested relationship was established randomly among neighbors in *Repre* while each parent in each local tree becomes the tester of its children in *Local*. As in Fig. 87, *d* of 1 and 7 and *d* of 3 and 5 consumed a similar amount of energy.

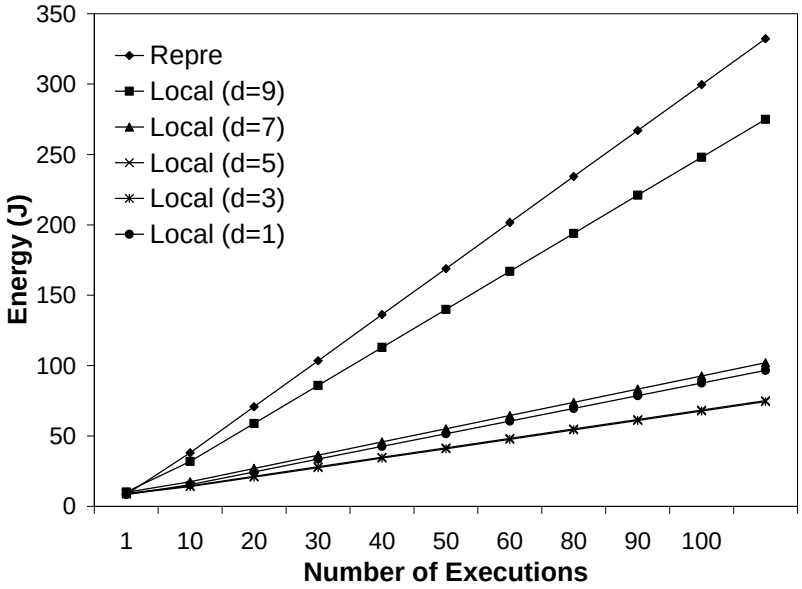


Fig. 88. Cumulative energy consumption for *Repre* and different number of local trees in *Local* (N=29524, a=10, r=3, 3 faults/exec.).

Fig. 89 shows the cumulative energy consumption for different number of executions from 1 to 100 for a network with 3280 nodes. *Local* uses less energy per diagnosis algorithm execution.

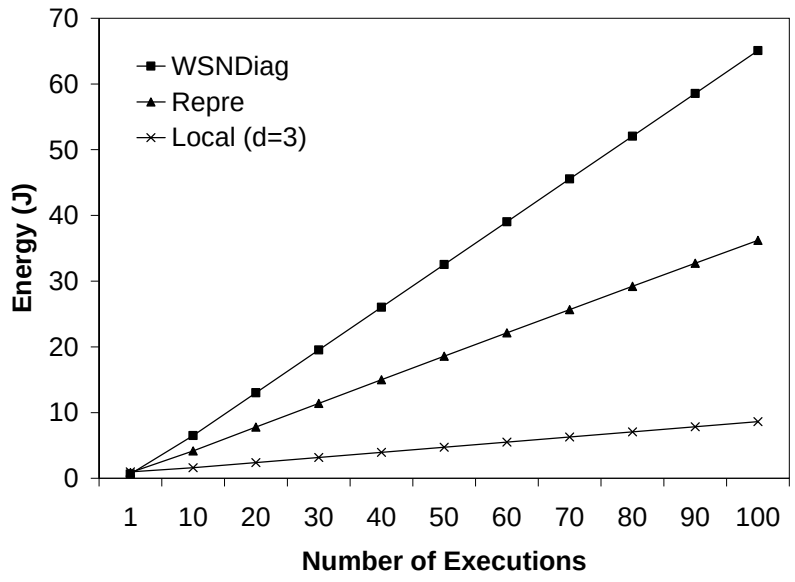


Fig. 89. Cumulative energy consumption in different approaches (N=3280, a=10, r=3, 3 faults/exec.).

Fig. 90 shows the difference between *WSNDiag*, *Repre*, and *Local* for one and ten executions for different network sizes. For more executions and larger network sizes, the energy consumption of *WSNDiag* becomes considerably larger than *Repre* and *Repre* becomes larger than *Local*. This is because *WSNDiag* completely rebuilds the communication tree each time, while *Repre* and *Local* only reconfigure the existing tree(s) as needed.

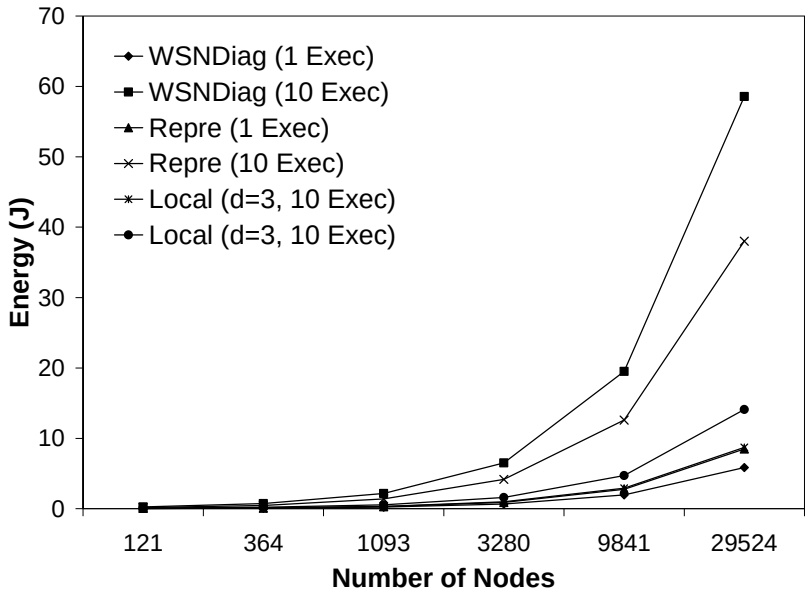


Fig. 90. Cumulative energy consumption in different approaches (a=10, r=3, 3 faults/exec.).

Fig. 91 shows the cumulative energy consumption for different node types in *Repre* and *Local* for different numbers of executions. The average value among all nodes of each type is shown. Leaf nodes in both *Repre* and *Local* always consume less energy per algorithm execution. In both approaches, the internal nodes consume more energy than the representative node(s) for a small number of executions due to the energy used by these nodes when building the initial tree. In later executions the representative node(s) consumes more energy disseminating diagnosis information.

In *Local*, the difference of energy consumption between local root and internal node (inter) is mainly due to summarized information shared among other local roots. Each local tree in *Local* has smaller size than a tree in *Repre*. Thus the energy consumption difference between local root and internal node is smaller than the difference between a representative and internal node in *Repre*. It means the representative overhead is

decreased in *Local*.

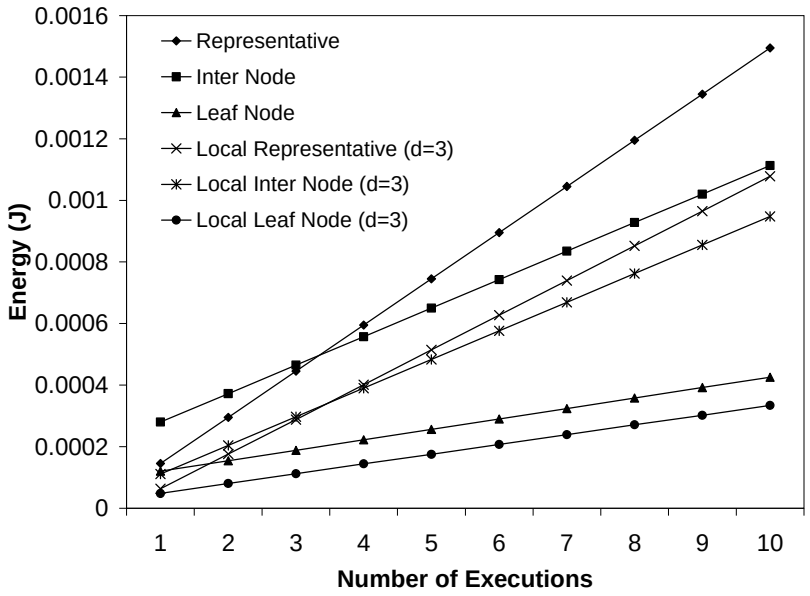


Fig. 91. Cumulative energy consumption per node (N=29524, a=10, r=3, 3 faults/exec.).

VII. CONCLUSIONS AND FUTURE WORK

In this dissertation, I have described the *INP* algorithm that creates a tree routing structure and maintains it for wireless sensor networks. The routing paths connecting nodes to the root are locally reconfigured against crash faulty nodes when information is delivered from sensor nodes to the control observer. Energy efficiency and scalability are provided for the reconfigurations by using only locally available relational information among neighbor nodes that does not need global maintenance throughout the tree. I have also described how *INP* can be extended for partition detection and how it can be used for reconnecting the partitioned region to the tree if new deployed nodes create joining paths.

INP was compared to *SWR* [18] through analytical calculations and ns-2 simulations. In densely deployed networks (i.e. nodes have many neighbors), most reconfiguration situations can by handled by *INP* with one of the five low-energy cases. In sparsely deployed networks (i.e. nodes have few neighbors), *INP* is able to reroute more cases than *SWR* by trying several different reconfiguration steps. Thus, *INP* had a higher delivery ratio, but higher latency than *SWR*.

The simulation results showed that each node in *INP* used a little less energy than in *SWR*. In *INP*, only the nodes related to a failed node participated in a reconfiguration and thus other nodes did not waste their energy. But in *SWR*, all neighbors are involved in the reconfiguration, even though they are not all necessary. Another reason than *INP* used less energy than *SWR* is that in most realistic cases a faulty node has several children, and these children eventually all have data messages to send to the root. *INP*

uses less energy for reconfiguration than *SWR* because the siblings located inside of the transmission range of a node that is looking for a new parent can get their new parents without their own efforts.

In *SWR*, to reduce communication overhead, updating the global value was postponed until a node received any information from its parent, on which the parent's global value piggybacked. If a node became involved in another reconfiguration before updating its global value, a loop could occur. This could result in increased latency (to detect and repair the loop) or cause message delivery failure.

INP was also compared by simulation to *GRAB-F*, a fixed transmission range version of *GRAB* [19]. *GRAB-F* had much lower delivery ratios and higher energy consumption than *INP* and *SWR*. This is because the broadcasting communication method used in *GRAB-F* was not as reliable as the unicast method used in *INP* and *SWR*. In *GRAB-F*, the number of forwarding paths could not be adequately limited, since a fixed transmission range was used, as in *INP* and *SWR*. In a dense network with high traffic, the fixed transmission range results in high message redundancy that causes a lot of message dropping by collisions and excess energy consumption.

In this dissertation, I have also described two new sensor-initiated crash fault diagnosis algorithms for wireless sensor networks, *Repre* and *Local*. *Local* was extended from *Repre* to provide scalability when the network size grows. Since *INP* was used for path reconfiguration, both static and dynamic faults were detected and reported to the control observer.

Repre was compared to *WSNDiag* [4] and *Local* was compared to *Repre* and

WSNDiag through analytical computations. *Repre* and *Local* use reliable one-to-one testing while *WSNDiag* use one-to-many testing. Using *INP*, *Repre* and *Local* maintained the tree communication structure by locally reconfiguring it as needed while *WSNDiag* made a tree per each use (diagnosis). Thus, the more reconfigurations and the larger the network, the larger the energy gap between *WSNDiag* and *Repre*, and between *Repre* and *Local*.

Several extensions to this research should be considered in future work. These include:

- Five basic reconfiguration cases in included in the current *INP* algorithm, related to the grandparent of the node looking for a new parent. If each node keeps track of *K* levels of ancestors and all related cases are used for reconfiguration at the first step, a new parent may be found before resorting to the more expensive search steps, such as *PFIND*. This would increase the reconfiguration success rate with lower latency, but with more messages during reconfiguration to update the relational information. The trade-offs between reconfiguration energy and knowledge maintenance energy must be studied.
- Keeping track of neighbors within *K* hops. This requires more messages during tree creation and maintenance, but can significantly reduce the chances of having only *UNKNOWN* neighbors. The trade-offs to identify the lowest-energy solution in different situations should be quantified.
- The current *INP* algorithm handles reconfigurations occurred when nodes send information to the root. Reconfigurations that occur when sending a message from parent to children should be studied. One solution is that when a parent

detects that one of its children is dead, it can inform the other children, who can attempt to connect to the children of the dead child. One option is for the parent to keep track of the grandchildren information. When reconfiguration is needed, the parent passes the children information of the dead child to its children and lets them find those grandchildren. Then the grandchildren know of the faulty parent and find their new parents using the current *INP* approach.

- Link failures were not considered in this research. If they can occur, then it will no longer be the case that because one node declares a neighbor faulty that other neighbors should trust that. Changing or extending *INP* to include link failures should be studied.
- Fixed transmission ranges were assumed in *INP*. Varying transmission ranges by adjusting transmission power based on current network density should be studied. Since power amplifier efficiency falls sharply with lower transmission power, variable power amplifiers do not make sense. Instead, a practical implementation would use discrete power levels with transmitters optimized for each level.
- In the current *Local* algorithm, the issue of local root reliability was not addressed. But it must be addressed in the future particularly when local roots are chosen from homogeneous failure prone sensor nodes. Having a backup node would be one solution.
- In the tree structure, node energy level is different depending on the location within the tree. The root node always consumes more energy than either leaf or

internal nodes, since all information ultimately arrives at the root node. For node energy balancing, a node with low energy levels should move lower in the tree, so that less traffic passes through it. One solution is for the node that has the highest energy level among the possible neighbors to become the new parent during reconfiguration. For this, nodes must share their energy levels during reconfiguration. Another solution is that if a lower energy level node has a higher energy level sibling node; it introduces some of its children to the sibling. And the sibling becomes a parent of those children. Further solutions include having the lower energy node limiting its number of children or giving up the current parent position.

- *Repre* and *Local* should be simulated and evaluated.

REFERENCES

[1] B. R. Badrinath and M. Srivastava, “Smart Space and Environment,” *IEEE Personal Communications*, vol. 7, no. 5, pp. 3-3, October 2000.

[2] C. Hsin and M. Liu, “A Distributed Monitoring Mechanism for Wireless Sensor Networks,” *Proc. 3rd ACM Workshop on Wireless Security (WISE),* pp. 57-66, September 2002.

[3] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, “Next Century Challenges: Scalable Coordination in Sensor Networks,” *Proc. 5th Annual ACM/IEEE International Conference on Mobile Computing and Networks (MobiCom 99),* pp. 263-270, August 1999.

[4] S. Chessa and P.Santi, “Crash Faults Identification in Wireless Sensor Networks,” *Computer Communications*, vol. 25, no. 14, pp. 1273-1282, September 2002.

[5] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, “Wireless Sensor Networks for Habitat Monitoring,” *MobiCom Workshop on Wireless Sensor Networks and Applications (WSNA*), pp. 122-131, September 2002.

[6] Y. Zhao, R. Govindan, and D. Estrin, “Residual Energy Scans for Monitoring Wireless Sensor Networks,” *IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 356 -362, March 2002.

[7] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, “A Taxonomy of Wireless Micro-Sensor Network Models,” *ACM Mobile Computing and Communications Review*, vol. 6, no. 2, pp.28-36, April 2002.

[8] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, "Protocols for Self-Organization of a Wireless Sensor Network," *IEEE Personal Communication*, vol. 7, no. 5, pp 16-27, October 2000.

[9] S. Rangarajan, A. T. Dahbura, and E. A. Ziegler, "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 312-333, February 1995.

[10] J. Pottie and W. J. Kaiser, "Embedding the Internet Wireless Integrated Network Sensors," *Communications of the ACM*, vol. 43, no. 5, pp. 51-58, May 2000.

[11] W. Ye, J. Heidemann, and D. Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks," *Proc. IEEE INFOCOM 2002*, pp. 1567-1576, June 2002.

[12] J. Kulik, W. R. Heinzelman, and H. Balakrishnan, "Negotiation-based Protocols for Disseminating Information in Wireless Sensor Networks," *Wireless Networks*, vol. 8, No. 2-3, pp. 169-185, March 2002.

[13] F. Akyildiz, W. Su, Y, Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications*, pp. 102-114, August 2002.

[14] C. Zhou and B. Krishnamachari, "Localized Topology Generation Mechanisms for Wireless Sensor Networks," *IEEE Global Telecommunications Conference (GLOBECOM),* vol. 3, pp. 1269-1273, December 2003.

[15] W. Zhang, G. Cao, and T. L. Porta, "Dynamic Proxy Tree-Based Dissemination Schemes for Wireless Sensor Networks," *IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, pp. 21-30, October 2004.

[16] J. Staddon, D. Balfanz, and G. Durfee, "Efficient Tracing of Failed Nodes in Sensor Networks," *MobiCom Workshop on Wireless Sensor Networks and Applications (WSNA),* pp. 122-131, September 2002.

[17] V. D. Park and M. S. Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *Proc. IEEE INFOCOM '97*, pp. 7-11, April 1997.

[18] D. Tian and N. D. Georganas, "Energy Efficient Routing with Guaranteed Delivery in Wireless Sensor Networks," *IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 3, pp.1923-1929, March 2003.

[19] F. Ye, G. Zhong, S. Lu, and L. Zhang, "Gradient Broadcast: A Robust Data Delivery Protocol for Large Scale Sensor Networks," *ACM Wireless Networks (WINET),* vol. 11, no. 3, pp. 285-298, May 2005.

[20] A. Juttner and A. Magi, "Tree Based Broadcast in Ad Hoc Networks," *Mobile Networks and Applications,* vol. 10, no. 5, pp. 753-762, October 2005.

[21] D. Braginsky and D. Estrin, "Rumor Routing Algorithm for Sensor Networks," *First ACM Workshop on Wireless Sensor Networks and Applications*, pp. 22-29, September 2002.

[22] C. Intagogonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *Proc. 6th Annual International Conference on Mobile Computing and Networking (MobiCom 2000)*, pp. 56-67, August 2000.

[23] J. N. Al-Karaki and A. E. Kamal, "Routing Techniques in Wireless Sensor Networks: A Survey," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 6-28, 2004.

[24] R. C. Shah and H. M. Rabaey, "Energy Aware Routing for Low Energy Ad Hoc Sensor Networks," *IEEE Wireless Communications and Networking Conference (WCNC),* vol. 1, pp. 350-355, March 2002.

[25] D. Ganesan, R. Govindan, R. Shenker, and D. Estrin, "Highly-Resilient, Energy-Efficient Multipath Routing in Wireless Sensor Networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 4, pp. 11-25, October 2001.

[26] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, pp. 153-181, Kluwer Academic Publishers, Norwell, MA, 1996.

[27] R. D. Poor, "Gradient Routing in Ad Hoc Networks," MIT Media Laboratory, http://www.media.mit.edu/pia/Research/ESP/texts/poorieeepaper.pdf, 2000.

[28] K. Ravindran, G. Singh, and P. Gupta, "Reconfiguration of Spanning Trees in Networks in the Presence of Node Failures," *Proc. 13th International Conference on Distributed Computing Systems,* pp. 219-226, May 1993.

[29] J. v. Greunen and J. Rabaey, "Lightweight Time Synchronization for Sensor Networks," *WSNA '03,* pp. 11-19, 2003.

[30] P. M. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths," *SIAM J. on Computing*, vol. 4, no. 3, pp. 375-380, September 1975.

[31] D. Eppstein, G. F. Italiano, R. Tamassia, and R. E. Tarjan, J. Westbrook and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph," *J. of Algorithms*, vol. 13, no. 1, pp. 33-54, March 1992.

[32] R. J. Wilson, *Introduction to Graph Theory, 3rd ed.*, Longman Scientific & Technical, New York, 1985.

[33] S. Kutten and A. Porat, "Maintenance of a Spanning Tree in Dynamic Networks," *Proc. 13th International Symposium on Distributed Computing,* pp. 342-355, 1999.

[34] F. C. Gartner, "A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms," *EPFL Tech. Rep. IC/2003/38*, pp. 1-12, June 2003.

[35] A. J. Mooij, N. Goga, and W. Wesselink, "A Distributed Spanning Tree Algorithm for Topology-Aware Networks," *Conference on Design, Analysis and Simulation of Distributed Systems (DASD '04),* pp. 169 - 178, 2004.

[36] A. Arora and A. Singhai, "Optimal, Nonmasking Fault-Tolerant Reconfiguration of Trees and Rings," Ohio State University*, Tech. Rep. CISRC-TR09*, 1994.

[37] E. Gafni and D. Bertsekas, "Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology," *IEEE Transactions on Communications*, vol. 29, no. 1, pp. 11-18, January 1981.

[38] N. Malpani, J. L. Welch, and N. Vaidya, "Leader Election Algorithms for Mobile Ad Hoc Networks," *4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communication*, pp. 96 – 103, August 2000.

[39] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Std. 802.11, 1999 edition.

[40] E. Jung, and D. M. H. Walker, "Reliable Energy Efficient Routing in Wireless Sensor Networks," *IEEE International Workshop on Resource Provisioning and Management in Sensor Networks (RPMSN),* November 2005.

[41] IETF Manet working group AODV draft, http://www3.ietf.org/proceedings/02jul/I-D/draft-ietf-manet-aodv-11.txt/, July 2002.

[42] L. M. Feeney and M. Nilsson, "Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment," *Proc. IEEE INFOCOM '2001*, vol. 3, pp. 1548-1557, 2001.

[43] W. Ye, J. Heidemann, and D. Estrin, "Medium Access Control with Coordinated, Adaptive Sleeping for Wireless Sensor Networks," *Tech. Rep. ISI-TR-567*, USC Information Sciences Institute, January 2003.

[44] Nrlsensorsim, http://cs.itd.nrl.navy.mil/work/sensorsim/, April 2004.

[45] UCB/LBNL/VINT Network Simulator – ns-2.27, http://www.isi.edu/nsnam/ns/, January 2005.

[46] A. Bagchi and S. L. Hakimi, "An Optimal Algorithm for Distributed System Level Diagnosis," *21st International Symposium on Fault-Tolerant Computing*, pp. 214-221, June 1991.

[47] M. J. Bearden and R. P. Bianchini, Jr., "Efficient and Fault-Tolerant Distributed Host Monitoring Using System-Level Diagnosis," *Proc. IFIP/IEEE International Conference on Distributed Platforms: Client/Server and Beyond*, pp. 159-172, February 1996.

[48] R. P. Bianchini, Jr. and R. W. Buskens, “Implementation of On-Line Distributed System-Level Diagnosis Theory,” *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 616-626, May 1992.

[49] E. P. Duarte Jr. and T. Nanya, “A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm,” *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 34-45, January 1998.

[50] A. K. Somani, “System Level Diagnosis: A Review,” *Tech. Rep.,* Dependable Computing Laboratory, Iowa State University, Ames, IA, 1997.

[51] F. P. Preparata, G. Metze, and R. T. Chien, “On the Connection Assignment Problem of Diagnosable Systems,” *IEEE Transactions On Computers*, vol. ec-16, no. 6, pp. 848-854, December 1967.

[52] S. Chutani and K. Vijayananda, “On the Suitability of the OSI Standard to the Diagnosis of Communication Networks,” *IEEE Catalogue no. 95TH8061*, pp. 116-120, 1995.

[53] A. T. Dahbura, K. K. Sabnani, and L. L. King, “The Comparison Approach to Multiprocessor Fault Diagnosis,” *IEEE Transactions on Computers*, vol. c-36, no. 3, pp. 373-378, March 1987.

[54] A. Sengupta and A. T. Dahbura, “On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach,” *IEEE Transactions on Computers,* vol. 41, no. 11, pp.1386-1396, November 1992.

[55] J. G. Kuhl and S. M. Reddy, "Fault-Diagnosis in Fully Distributed Systems," *11st International Symposium on Fault-Tolerant Computing (FTCS-11)*, pp.100-105, 1981.

[56] S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, "A Diagnosis Algorithm for Distributed Computing Systems with Failure and Repair," *IEEE Transactions on Computers*, vol. c-33, no. 3, pp. 223-233, March 1984.

[57] R. Bianchini, K. Goodwin, and D. S. Nydick, "Practical Application and Implementation of System-Level Diagnosis Theory," *20st International Symposium on Fault-Tolerant Computing (FTCS-20),* pp.332-339, June 1990.

[58] R. Bianchini and R. Buskens, "An Adaptive Distributed System-Level Diagnosis Algorithm and Its Implementation," Proc. 21st Int. Symp. Fault-Tolerant Computing, pp. 222-229, June 1991.

[59] S. L. Hakimi and K. Nakajima, "On Adaptive System Diagnosis," *IEEE Transactions on Computers*, vol. c-33, no. 3, pp. 234-240, 1984.

[60] M. Barborak and M. Malek, "The Consensus Problem in Fault-Tolerant Computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171-220, 1993.

[61] M. Stahl, R. Buskens, and R. Bianchini, "Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks," *Proc. IEEE 11th Symposium Reliable Distributed Systems*, pp. 180-187, October 1992.

[62] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Sensor Information Networking Architecture and Applications," vol. 8, no. 4, *IEEE Personal Communication Magazine*, pp. 52-59, August 2001.

[63] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, “Macaw: A Media Access Protocol for Wireless LAN’s,” *Proc. of the ACM SIGCOM*, pp. 212-225, September 1994.

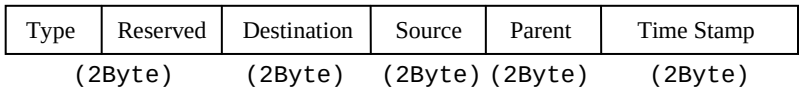
[64] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms (Second Edition)*, The MIT Press. Cambridge, MA, 2001.

[65] W. Ye, J. Heidemann, and D. Estrin, “A Flexible and Reliable Radio Communication Stack on Motes,” *Tech. Rep. ISI-TR-565*, USC Information Sciences Institute, September 2002.

APPENDIX A MESSAGE FORMATS

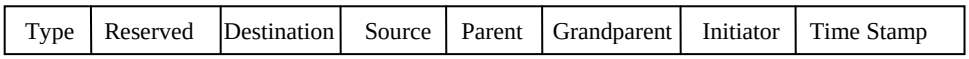
The following is the list of messages used for the *INP* reconfiguration algorithm and their function and formats. Message formats are explained based on the IEEE 802.11 MAC format since simulations are done with this. For computational analysis, the S-MAC format was used and the format is explained below when additional messages for the diagnosis algorithms are introduced.

PARENT: It is used for making the initial tree. By exchanging this information with other nodes, each node gets its parent, grandparent, children, and siblings.




```
struct hdr_Parent {
        u_int8_t        Parent_type;         // Packet Type
        u_int8_t        reserved[3];         //
        nsaddr_t        Parent_dst;          // Destination Node
        nsaddr_t        Parent_src;          // Source Node
        nsaddr_t        Parent_parent;       // Parent Node
        double          Parent_timestamp;    // when Parent sent
                                               // for computing latency
        inline int size() {
                int sz = 5*sizeof(u_int32_t);
                assert (sz >= 0);
                return sz;
       }
};
```

INP ("I need parent"): It is used for a node to find a new parent in the tree.




```
struct hdr_INP {
        u_int8_t        INPtype;               // Packet Type
        u_int8_t        reserved[2];           //
        nsaddr_t        INP_dst;               // Destination Node
        nsaddr_t        INP_src;               // Source Node
        nsaddr_t        INP_parent;            // Parent Node
        nsaddr_t        INP_grandparent;       // Grandparent Node
        nsaddr_t        INP_initiator;         // Initiator Node
```

```
        double          INP_timestamp;          // when INP sent
                                                // to compute route
                                                // discovery latency
        inline int size() {
               int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACKINP("Acknowledgement of INP"): It is used when a node can become a new parent of the *INP* sender.

Type	Case Num	Reserved	Destination	Source	Parent	Initiator	Time Stamp
	(2Byte)		(2Byte)	(2Byte)	(2Byte)	(2Byte)	(2Byte)

```
struct hdr_ACKINP {
       u_int8_t        ACKINP_type;       // Packet Type
       u_int8_t        ACKINP_caseNum     // Cycle Free Path Type
                                          // (Case 1 to 5)
       u_int8_t        reserved[2];       //
       nsaddr_t        ACKINP_dst;        // Destination Node
       nsaddr_t        ACKINP_src;        // Source Node
       nsaddr_t        ACKINP_parent;     // Parent Node
       nsaddr_t        ACKINP_initiator;  // Initiator Node
       double          ACKINP_timestamp

       inline int size() {
              int sz = 6*sizeof(u_int32_t);
              assert (sz >= 0);
              return sz;
       }
};
```

ACPINP("Acceptance of INP"): It is used when an *INP* sender declares one of *ACKINP* senders as its new parent.

Type	Case Num	Reserved	Destination	Source	Parent	Grandparent	Fault Node	Time Stamp

```
struct hdr_ACPINP {
        u_int8_t        ACPINP_type;         // Packet Type
        u_int8_t        ACPINP_caseNum;      // case Number
        u_int8_t        reserved[2];
        nsaddr_t        ACPINP_dst;          // Destination Node
```

```
        nsaddr_t        ACPINP_src;          // Source Node
        nsaddr_t        ACPINP_parent;       // Parent Node
        nsaddr_t        ACPINP_gp;           // Grandparent Node
        nsaddr_t        ACPINP_faultNode;    // Fault Node
        double          ACPINP_timestamp;

        inline int size() {
               int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

CNFCF ("Cannot find cycle free"): It is used when an *INP* sender is looking for its siblings' helps when it did not receive a *ACKINP*.

Type	Reserved	Destination	Source	Parent	Grandparent	Initiator	Time Stamp

```
struct hdr_CNFCF {
        u_int8_t        CNFCF_type;         // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        CNFCF_dst;          // Destination Node
        nsaddr_t        CNFCF_src;          // Source Node
        nsaddr_t        CNFCF_parent;       // Parent Node
        nsaddr_t        CNFCF_grandparent;  // Grandparent Node
        nsaddr_t        CNFCF_initiator;    // Initiator Node
        double          CNFCF_timestamp;

        inline int size() {
               int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACKCNFCF ("Acknowledgement of Cannot find cycle free"): It is used when a sibling node of the initiator can become a new parent since it received an *ACKINP* from one of its neighbors.

Type	Reserved	Destination	Source	Finder	Finder's Parent	Initiator	Time Stamp

```
struct hdr_ACKCNFCF {
        u_int8_t        ACKCNFCF_type;         // Packet Type
        u_int8_t        reserved[3];
```

```
        nsaddr_t        ACKCNFCF_dst;          // Destination Node
        nsaddr_t        ACKCNFCF_src;          // Source Node
        nsaddr_t        ACKCNFCF_finder;       // Finder Node
        nsaddr_t        ACKCNFCF_finderP;      // Finder's parent
        nsaddr_t        ACKCNFCF_initiator;    // Initiator Node
        double          ACKCNFCF_timestamp;

        inline int size() {
               int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACPCNFCF ("Acceptance of CNFCF"): It is used when the *CNFCF* sender declares one of the *ACKCNFCF* senders as its new parent.

Type	Case Num	Reserved	Destination	Source	Parent	Finder	Finder's Parent	Time Stamp

```
struct hdr_ACPCNFCF {
        u_int8_t        ACPCNFCF_type;         // Packet Type
        u_int8_t        reserved[2];           //
        u_int8_t        ACPCNFCF_caseNum;      // case numher
        nsaddr_t        ACPCNFCF_dst;          // Destination Node
        nsaddr_t        ACPCNFCF_src;          // Source Node
        nsaddr_t        ACPCNFCF_parent;       // New Parent
        nsaddr_t        ACPCNFCF_finder;       // Finder
        nsaddr_t        ACPCNFCF_finderP;      // Finder's Parent
        double          ACPCNFCF_timestamp;    // when ACPICNYP sent;

        inline int size() {
        int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
        return sz;
        }
};
```

ICNYP ("I cannot be your parent"): It is used when a node is looking for its children's help. It checks whether its child becomes its new parent using this message, when it did not receive any *ACKINP* or *ACKCNFCF* messages.

Type	Reserved	Destination	Source	Parent	Grandparent	Initiator	Time Stamp

```
struct hdr_ICNYP {
        u_int8_t        ICNYP_type;             // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        ICNYP_dst;              // Destination Node
        nsaddr_t        ICNYP_src;              // Source Node
        nsaddr_t        ICNYP_parent;           // Parent Node
        nsaddr_t        ICNYP_grandparent;      // Grandparent Node
        nsaddr_t        ICNYP_initiator;        // Initiator Node
        double          ICNYP_timestamp;

        inline int size() {
               int sz = 7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
       }
};
```

ACKICNYP ("Acknowledgement of ICNYP"): It is used when a child node of the initiator can become a new parent since it received an *ACKINP* from one of its neighbors.

Type	Case Num	Reserved	Destination	Source	Finder	Finder's Parent	Initiator	Time Stamp

```
struct hdr_ACKICNYP {
        u_int8_t        ACKICNYP_type;        // Packet Type
        u_int8_t        ACKICNYP_caseNum;     // Case number
        u_int8_t        reserved[2];
        nsaddr_t        ACKICNYP_dst;         // Destination Node
        nsaddr_t        ACKICNYP_src;         // Source Node
        nsaddr_t        ACKICNYP_finder;      // Node which finds new path
        nsaddr_t        ACKICNYP_finderP;     // Finder's parent
        nsaddr_t        ACKICNYP_initiator;   // Initiator Node
        double          ACKICNYP_timestamp;

        inline int size() {
               int sz =7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACPICNYP ("Acceptance of ICNYP"): It is used when the *ICNYP* sender declares one of the *ACKICNYP* senders as its new parent. The format is same as *ACPCNFCF*.

INI ("I need information"): It is used when a node needs neighbors' information before using the *PFIND* message.

Type	Reserved	Destination	Source	Fault Node	Time Stamp

```
struct hdr_INI {
        u_int8_t        INI_type;         // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        INI_dst;          // Destination Node
        nsaddr_t        INI_src;          // Node itself
        nsaddr_t        INI_faultNode;    // Fault Node
        double          INI_timestamp;

        inline int size() {
               int sz = 5*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACKINI ("Acknowledgement of INI"): It is used when a neighbor send its information to the *INI* sender.

Type	Reserved	Destination	Source	Parent	Grandparent	Time Stamp

```
struct hdr_ACKINI {
        u_int8_t        ACKINI_type;         // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        ACKINI_dst;          // Destination: INI sender
        nsaddr_t        ACKINI_src;          // Node itself
        nsaddr_t        ACKINI_parent;       // Parent Node
        nsaddr_t        ACKINI_gp;           // Grandparent Node
        double          ACKINI_timestamp;

        inline int size() {
               int sz = 6*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

PFIND ("Path find"): It is used when an *UNKNOWN* neighbor is selected and checked whether it becomes a new parent or not.

Type	Reserved	Destination	Sender	Relayer	Parent	Grandparent	Time Stamp

```
struct hdr_PFIND {
        u_int8_t        PFIND_type;      // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        PFIND_dst;       // Destination Node:
                                         // selected neighbor
        nsaddr_t        PFIND_src;       // PFIND sender
        nsaddr_t        PFIND_relayer;   // Relay Node
        nsaddr_t        PFIND_parent;    // Parent Node of PFIND sender
        nsaddr_t        PFIND_gp;        // Grandparent of PFIND sender
        double          PFIND_timestamp;

         inline int size() {
               int sz =7*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
        }
};
```

ACKPFIND ("Acknowledgement of PFIND"): It is used when a relay node that receives *PFIND* finds a new path and lets the sender of *PFIND* know it along the reverse path direction.

Type	Reserved	Destination	Source	Origin	Parent	Time Stamp

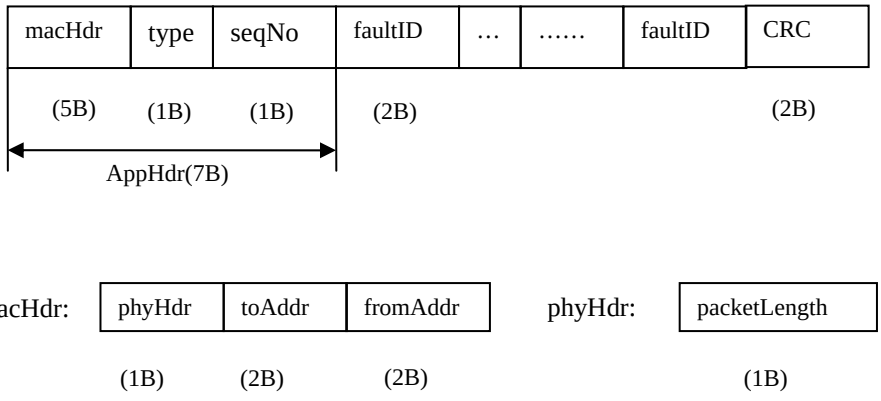
```
struct hdr_ACKPFIND {
        u_int8_t        ACKPFIND_type;        // Packet Type
        u_int8_t        reserved[3];
        nsaddr_t        ACKPFIND_dst;         // Destination Node:
                                              // previous relayer
        nsaddr_t        ACKPFIND_src;         // Source Node
        nsaddr_t        ACKPFIND_origin;      // Sender of PFIND
        nsaddr_t        ACKPFIND_parent;      // Parent Node
        double          ACKPFIND_timestamp;

        inline int size() {
               int sz = 6*sizeof(u_int32_t);
               assert (sz >= 0);
               return sz;
         }
};
```

ACPPFIND ("Acceptance of PFIND"): It is used when the *PFIND* sender declares one of the *ACKPFIND* senders as its new parent. The format is same with *ACPINP*.

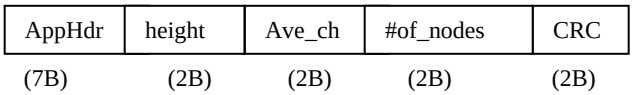
The following is the list of additional messages that are used for the *Repre and Local* diagnosis algorithms. For measuring the energy consumption of those algorithms and alternatives, packet formats followed the radio communication stack introduced in [65]. This stack is used on the Mica Motes developed at USC/ISI and UCLA [65]. This stack includes sensor-MAC (i.e., S-MAC), [11][42][65] a medium-access control protocol that was specially designed for wireless sensor networks. The unique two-byte transmitter and receiver addresses are in the MAC header (macHdr) with a one-byte Physical header (phyHdr) [65].

INFO ("Diagnosis Information"): It is a diagnosis message disseminated among the nodes.

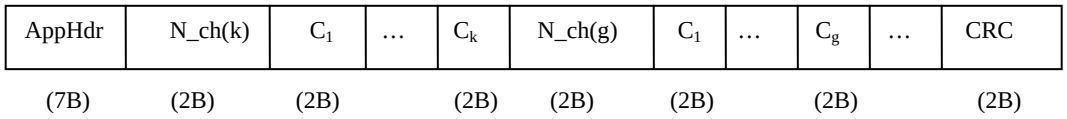


NRH ("Nodes_Children_Height"): It describes the total number of descendants, average number of children per each parent, and the height of each node. Each *NRH*

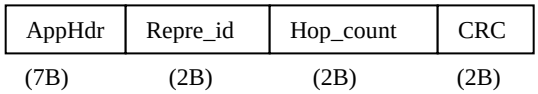
message initiated from each leaf is aggregated and updated in each intermediate node and arrives at the initial representative node. The control observer uses this message for computing hop count between a local root and the next local root.



CHILDINFO ("Child information"): It is the children information that each intermediate node sends to its parent after establishing a tree or local tree(s).



HOP: It is initiated from the initial representative node and used for determining and announcing each local root to the nodes. It has each local root node ID and the hop count.



IAD ("I am dying"): It is used for a node that will die due to battery depletion to give early warning to the nodes tested by that node. Then each receiving node(s) can know that its tester node does not have enough power to complete the diagnosis procedure and they regard the tester as faulty and each selects another node as its tester. This message is also used for other purposes. In the *INP* algorithm, it helps the neighbors to get the relational information and use it for path reconfiguration.

AppHdr	myID.p	myID.gp	Testedby	Testerof	CRC
(7B)	(2B)	(2B)	(2B)	(2B)	(2B)

For each message type below, this control packet is used.

AppHdr	CRC
(7B)	(2B)

IMA ("I am alive"): It is used when each tested node reports its health to its tester.

TEST: It is used when a tester node tests its tested node.

REPLY: It is used when a fault free tested node tells its health to its tester after receiving a *TEST* message.

TESTME ("Test me"): It is used for a node that needs a tester for itself. Whenever a node learns that its tester node is faulty (or will become faulty soon), the node sends this message to its neighbors.

ACKTM ("Acknowledgement of TESTME"): It is an acknowledgement of the *TESTME* message. It is sent by each fault-free neighbor that receives *TESTME*.

ACPTM ("Acceptance of TESTME"): It is a confirmation message of *ACKTM*. When a node receives several *ACKTM* from fault-free neighbors, it chooses one as its tester and sends *ACPTM* to that node. So the node that receives *ACPTM* becomes its tester.

ACKINFO ("Acknowledgement of INFO"): It is an acknowledgement of *INFO*. Each node can detect the faulty status of a neighbor when it does not receive this message from the neighbor after sending an *INFO* message to it.

ACKNRH("Acknowledgement of NRH"): This is an acknowledgement of *NRH*.

ACKCHILDINFO ("Acknowledgement of CHILDINFO"): It is an acknowledgement of *CHILDINFO*. Each node can detect the faulty status of its parent node when it does not receive this message after sending *CHILDINFO*.

ACKHOP: This is an acknowledgement of ACKHOP.

APPENDIX B SIMULATION ENVIRONMENTS

To install NRLsensorsim for simulating sensor networks in the ns-2 network simulator (ns-2.27), nrlsensorsim-2.27.tgz was downloaded from http://downloads.pf.itd.nrl.navy.mil/archive/nrlsensorsim/.

For compatibility of NRLsensorsim with ns-2.27, a patch file (patch_script-2.27.sh) that modifies ns-2.27 must be run in nrlsensorsim-2.27 directory before installing of ns-2.27 as follows:

```
>tar -xzvf ns-allinone-2.27.tgz
>tar -xzvf nrlsensorsim-2.27.tgz
>cd nrlsensorsim-2.27/
>./patch_script-2.27.sh
>cd ../ns-allinone-2.27/
>./install
```

The modification details are described in http://downloads.pf.itd.nrl.navy.mil/archive/nrlsensorsim/INSTALL-2.27.txt.

A simulation of the *INP* routing protocol that includes creation of the routing tree and local reconfiguration against fault nodes was made in the ns-allinone-2.27/ns-2.27/INP directory. The following 9 files are under the INP directory;

```
INP/INP.cc
INP/INP.h
INP/INP.tcl
INP/INP_logs.cc
INP/INP_packet.h
INP/INP_rqueue.cc
INP/INP_rqueue.h
INP/INP_rtable.cc
INP/INP_rtable.h
```

For these files to be compiled, the following INP object files shown as bold are added into the *OBJ_CC* variable in the Makefile located in the upper directory (i.e., ns-

allinone-2.27/ns-2.27) ;

```
OBJ_CC = \
      …
      INP/INP_logs.o INP/INP.o \
      INP/INP_rtable.o INP/INP_rqueue.o \
      …
$(OBJ_STL)
```

For INP to be integrated with ns-2.27, a declaration of new INP packet type is included in common/packet.h as follows:

```
enum packet_t {
        …
        PT_INP,
        PT_NTYPE // This MUST be the LAST one
};
```

Also, a textual name for a new INP packet type is added into the constructor of **p_info** class in common/packet.h

```
p_info() {
        ...
        name_[PT_INP]= "inp";
}
```

To trace new INP packets when the packets are sent, received, and dropped, **format_INP()** function was added into **trace/cmu-trace.cc** and **trace/cmu-trace.h**

At **trace/cmu-trace.h**

```
class CMUTrace : public Trace {
        ...
private:
        ...
        void    format_INP(Packet *p, int offset);
};
```

At **trace/cmu-trace.cc**

```
...
#include <INP/INP_packet.h>
...
void
CMUTrace::format(Packet* p, const char *why)
{
        ...
      switch(ch->ptype()) {
         ...
         case PT_INP:
                     format_INP(p, offset);
                     break;
        default:
          ...
}

...
void
CMUTrace::format_INP(Packet *p, int offset)
{
   struct hdr_INP *ah = HDR_INP(p);
   struct hdr_ip *ih = HDR_IP(p);

   switch(ah->ah_type) {
      case INPTYPE_Parent:
      case INPTYPE_INP:
      case INPTYPE_CNFCF:
      case INPTYPE_ICNYP:
      case INPTYPE_INI:
      case INPTYPE_ACKICNYP:
      case INPTYPE_ACKINI:
      case INPTYPE_ACKINP:
      case INPTYPE_ACPINP:
      case INPTYPE_ACPICNYP:
      case INPTYPE_PFIND:
      case INPTYPE_ACKPFIND:
            if (pt_->tagged()) {
                  …
            } else if (newtrace_) {
                  …
            } else {

                  sprintf(pt_->buffer() + offset,
                    "[0x%x %d %d %d %f] (%s)",
                    rp->INP_type,
                    rp->INP_dst,
                    rp->INP_src,
                    rp->INP_parent,
                    rp->INP_timestamp,
                    rp->INP_type == INPTYPE_INP ? "INP" :
                    (rp->rp_type == INPTYPE_CNFCF ? "CNFCF" :
```

```
                …
            }
            break;
        default:
            #ifdef WIN32
                  fprintf(stderr,
                    "CMUTrace::format_INP: invalid INP packet type\n");
            #else
                  fprintf(stderr,
                    "%s: invalid INP packet type\n", __FUNCTION__);
            #endif
                abort();
        }
}
```

Tcl library files were modified to add the INP packet type (at **tcl/lib/ns-packet.tcl)**, to define default values for bound attributes (at **tcl/lib/ns-default.tcl**)**,** and to add the procedures that set the INP routing agent for a wireless node (at **tcl/lib/ns-lib.tcl**) like these;

At **tcl/lib/ns-packet.tcl**

```
foreach prot {
    INP
    AODV
    # ...
    NV
} {
    add-packet-header $prot
}
```

At **tcl/lib/ns-default.tcl**

```
# ...
# Defaults defined INP
Agent/INP set accessible_var_ true
```

At **tcl/lib/ns-lib.tcl**

```
 Simulator instproc create-wireless-node args {
      # ...
      switch -exact $routingAgent_ {
```

```
            INP {
                set ragent [$self create-INP-agent $node]
            }
            # ...
        }
        # ...
}

Simulator instproc create-INP-agent { node } {
         # create INP routing agent
         set ragent [new Agent/INP [$node node-addr]]
         $self at 0.0 "$ragent start"
         $node set ragent_ $ragent
         return $ragent
}
```

For INP packets to be treated as routing packets at the queue (i.e., priqueue) that considers routing packets with high priority packets, **queue/priqueue.cc** is modified as follows:

At **queue/priqueue.cc**

```
void
PriQueue::recv(Packet *p, Handler *h)
{
    struct hdr_cmn *ch = HDR_CMN(p);
       if (Prefer_Routing_Protocols) {
               switch(ch->ptype()) {
                       . . .
                       case PT_AODV:
                       case PT_INP:
                       default:
                              Queue::recv(p, h);
               }
       }
       else {
         Queue::recv(p, h);
       }
}
```

Before executing the ***make*** command, for **common/packet.cc** to be recompiled, the timestamp of **common/packet.cc** must be modified by using the UNIX ***touch*** command as follows. This is because **common/packet.h** is changed and **common/packet.cc** was

not changed.

```
> touch common/packet.cc
> make
```

VITA

Eun Jae Jung was born in Wonju, Korea. He received his Bachelor's degree in mathematics from Myongji University, Yongin, Korea in 1991, and earned his Master's degree in computer science from Oklahoma State University, Stillwater, in 1996. He received a Ph.D degree in computer science at Texas A&M University in December 2007, under the supervision of Dr. Duncan M. H. Walker. He worked in the Computer Science Department Computer Support Group as a graduate assistant, from January 1999 through December 2003. His current research interests include wireless sensor networks, routing protocols, and system level diagnosis. His permanent address is: 42-1 Bukmoon-Dong, Andong-Si, Kyeong-Sang-Buk-Do, Republic of Korea.