Murat Demirbas
Computer Science and Engineering Department, The Ohio State University

Anish Arora
Computer Science and Engineering Department, The Ohio State University

Mohamed Gouda
Computer Sciences Department, The University of Texas at Austin

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

p. cm.—(Wiley series in survey methodology)
"Wiley-Interscience."
Includes bibliographical references and index.

HA31.2.S873 2004
001.4'33—dc22 2004044064
Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CHAPTER 9

PURSUER-EVADER TRACKING IN SENSOR NETWORKS

9.1 ABSTRACT

In this paper we present a self-stabilizing program for solving a pursuer-evader problem in sensor networks. The program is a hybrid between two orthogonal programs, an evader-centric program and a pursuer-centric program, and can be tuned for tracking speed or energy efficiency. In the program, sensor nodes close to the evader dynamically maintain a tracking tree of depth R that is always rooted at the evader. The pursuer, on the other hand, searches the sensor network until it reaches the tracking tree, and then follows the tree to its root in order to catch the evader.

9.2 INTRODUCTION

Due to its importance in military contexts, pursuer-evader tracking has received significant attention [6, 8, 23, 25] and has been posed by the DARPA network embedded software technology (NEST) program as a challenge problem. Here, we consider the problem in the context of wireless sensor networks. Such networks comprising potentially many thousands of low-cost and low-power wireless sensor nodes have recently became feasible, thanks to advances in microelectromechanical systems technology, and are being regarded as a realistic basis for deploying large-scale pursuer evader tracking.

Previous work on the pursuer-evader problem is not directly applicable to tracking in sensor networks, since these networks introduce the following challenges: Firstly, sensor

nodes have very limited computational resources (e.g., 8K RAM and 128K flash memory); thus, centralized algorithms are not suitable for sensor networks due to their larger computational requirements. Secondly, sensor nodes are energy constrained; thus, algorithms that impose an excessive communication burden on nodes are not acceptable since they drain the battery power quickly. Thirdly, sensor networks are fault-prone: message losses and corruptions (due to fading, collusion, and hidden node effect), and node failures (due to crash and energy exhaustion) are the norm rather than the exception. Thus, sensor nodes can lose synchrony and their programs can reach arbitrary states [20]. Finally, on-site maintenance is not feasible; thus, sensor networks should be self-healing. Indeed, one of the emphases of the NEST program is to design low-cost fault-tolerant, and more specifically self-stabilizing, services for the sensor network domain.

In this paper we present a tunable and self-stabilizing program for solving a pursuer-evader problem in sensor networks. The goal of the pursuer is to catch the evader (despite the occurrence of faults) by means of information gathered by the sensor network. The pursuer can move faster than the evader. However, the evader is omniscient —it can see the state of the entire network— whereas the pursuer can only see the state of one sensor node (say the nearest one). This model captures a simple, abstract version of problems that arise in tracking via sensor networks.

Tunability. We achieve tunability of our program by constructing it to be a hybrid between two orthogonal programs: an evader-centric program and a pursuer-centric program.

In the evader-centric program, nodes communicate periodically with neighbors and dynamically maintain a tracking tree structure that is always rooted at the evader. The pursuer eventually catches the evader by following this tree structure to the root: the pursuer asks the closest sensor node who its parent is, then proceeds to that node, and thus, reaches the root node (and hence the evader) eventually.

In the pursuer-centric program, nodes communicate with neighbors only at the request of the pursuer: When the pursuer reaches a node, the node resets its recorded time of a detection of an evader to zero and directs the pursuer to a neighboring node with the highest recorded time.

The evader-centric program converges and tracks the evader faster, whereas the pursuer-centric program is more energy-efficient. In the hybrid program we combine the evader-centric and pursuer-centric programs:

1. We modify the evader-centric program to limit the tracking tree to a bounded depth R to save energy.

2. We modify the pursuer-centric program to exploit the tracking tree structure.

The hybrid program is tuned for tracking speed or energy efficiency by selecting R appropriately. In particular, for the extended hybrid program in Section 9.7, the tracking time is $3 * (D - R) + R * \alpha/(1 - \alpha)$ steps, and at most n communications take place at each program step, where D denotes the diameter of the network, α is the ratio of the speed of the evader to that of the pursuer, and n is the number of sensor nodes included in the tracking tree.

Self-stabilization. In the presence of faults, our program recovers from arbitrary states to states from where it correctly tracks the evader; this sort of fault-tolerance is commonly referred to as stabilizing fault-tolerance. In particular, starting from any arbitrary state, the tracking time is $2R + 3 * (D - R) + R * \alpha/(1 - \alpha)$ steps for our extended hybrid program.

Organization of the paper. After presenting the system and fault model in the next section, we present an evader-centric program in Section 9.4 and a pursuer-centric program in Section 9.5. In Section 9.6, we present the tunable, hybrid program combining the previous two programs. We present an efficient version of the hybrid program in Section 9.7. We present implementation and simulation results in Section 9.8. Finally, we discuss related work and make concluding remarks in Section 9.9.

9.3 THE PROBLEM

System model. A sensor network consists of a (potentially large) number of sensor nodes. Each node is capable of receiving/transmitting messages within its field of communication. All nodes within this communication field are its neighbors; we denote this set for node j as $nbr.j$. We assume the nbr relation is symmetric and induces a connected graph. (Protocols for maintaining biconnectivity in sensor networks are known [12, 26].)

Problem statement. Given are two distinguished processes, the pursuer and the evader, that each reside at some node in the sensor network. Each node can immediately detect whether the pursuer and/or the evader are resident at that node.

Both the pursuer and the evader are mobile: each can atomically move from one node to another, but the speed of evader movement is less than the speed of the pursuer movement.

The strategy of evader movement is unknown to the network. The strategy could in particular be intelligent, with the evader omnisciently inspecting the entire network to decide whether and where to move. By way of contrast, the pursuer strategy is based only on the state of the node at which it resides.

Required is to design a program for the nodes and the pursuer so that the pursuer can "catch" the evader, i.e., guarantee in every computation of the network that eventually both the pursuer and the evader reside at the same node.

Programming model. A program consists of a set of variables, node actions, pursuer actions, and evader actions.

Each variable and each action resides at some node. Variables of a node j can be updated only by j's node actions. Node actions can only read the variables of their node and the neighboring nodes. Pursuer actions can only read the variables of their node. The evader actions can read the variables of the entire program, however, they cannot update any of these variables.

Each action has the form:

$$\langle \text{guard} \rangle \longrightarrow \langle \text{assignment statement} \rangle$$

A guard is a boolean expression over variables. An assignment statement updates one or more variables.

A state is defined by a value for every variable in the program, chosen from the predefined domain of that variable. An action whose guard is true at some state is said to be *enabled* at that state.

We assume maximal parallelism in the execution of node actions. At each state, each node executes all actions that are enabled in that state. (Execution of multiple enabled actions in a node is treated as executing them in some sequential order.) Maximal parallelism is not assumed for the execution of the pursuer and evader actions. Recall, however, that the speed of execution of the former exceeds that of the latter. For ease of exposition, we assume that evader and pursuer actions do not occur strictly in parallel with node actions.

A computation of the program is a maximal sequence of program steps: in each step, actions that are enabled at the current state is executed according to the above operational semantics, thereby yielding the next state in the computation. The maximality of a computation implies that no computation is a proper prefix of another computation.

We assume that each node has a clock, that is synchronized with the clocks of other nodes. This assumption is reasonably implemented at least for Mica nodes [18]. Later, in Section 9.8, we show how our programs can be modified to work without the synchronized clocks assumption.

Notation. In this paper, we use j, k, and l to denote nodes. We use $var.j$ to denote the variable var residing at j. We use $\Box$ to separate the actions in a program and $x :\in A$ to denote that x is assigned to an element of set A.

Each parameter in a program ranges over the nbr set of a node. The function of a parameter is to define a set of actions as one parameterized action. For example, let k be a parameter whose value is 0, 1, or 2; then an action act of node j parameterized over k abbreviates the following set of actions:

$$act\backslash(k = 0) \quad \Box \quad act\backslash(k = 1) \quad \Box \quad act\backslash(k = 2)$$

where $act\backslash(k = i)$ is act with every occurrence of k substituted with i.

We describe certain conjuncts in a guard in English: $\{$Evader resides at $j\}$ and $\{$Evader detected at $j\}$. The former expression evaluates to true at all states where the evader is at j whereas the latter evaluates to true only at the state immediately following any step where the evader moves to node j, and evaluates to false in the subsequent states even if the evader is still at j.

We use N to denote the number of nodes in the sensor network, D the diameter of the network, and M the distance between the pursuer and evader. We use distance to refer to hop-distance in the network; unit distance is the single-hop communication radius. Finally, we use α to denote the ratio of the speed of the evader to that of the pursuer.

Evader action. In each of the programs that we present in this paper, we use the following evader action.

$$\{ \text{ Evader resides at } j \ \} \longrightarrow \text{ Evader moves to } l,\ l :\in \{k \mid k \in nbr.j \cup \{j\}\}$$

When this action is executed, the evader moves to an arbitrary neighbor of j or skips a move. This notion of nondeterministic moves suffices to capture the strategy of an omniscient evader.

Recall from the discussion in the problem statement that, when the evader moves to a node, the node immediately detects this fact (i.e., the detection actions have priority over normal node actions and are fired instantaneously).

Fault model. Transient faults may corrupt the program state. Transient faults may also fail-stop or restart nodes (in a manner that is detectable to their neighbors); we assume that the connectivity of the graph is maintained despite these faults.

A program P is stabilizing fault-tolerant iff starting from an arbitrary state, provided that no other faults occur during recovery, P eventually recovers to a state from where its specification is satisfied.

9.4 AN EVADER-CENTRIC PROGRAM

In this section we present an evader-centric solution to the pursuer-evader problem in sensor networks. In our program every sensor node, j, maintains a value, $ts.j$, that denotes the latest timestamp that j knows for the detection of the evader. Initially, for all j, $ts.j = 0$. If j detects the evader, it sets $ts.j$ to its clock's value. Every node j periodically updates its $ts.j$ value based on the ts values of its neighbors: j assigns the maximum timestamp value it is aware of as $ts.j$. We use $p.j$ (read parent of j) to record the node that j received the maximum timestamp value. Initially, for all j, $p.j$ is set to null, i.e., $p.j = \perp$. As the information regarding the evader propagates through the network via gossipping of the neighbors, the parent variables at these nodes are set accordingly. Note that the parent relation embeds a tree rooted at the evader on the sensor network. We refer to this tree as the *tracking tree*.

In addition to above variables, we maintain a variable $d.j$ at each node j. Initially, for all j, $d.j = \infty$. When the evader is detected at a node j, $d.j$ is set to 0. Otherwise, $d.j$ is updated by setting it to be the parent's d value plus 1, i.e., $d.j := d.(p.j) + 1$. This way, $d.j$ at each node corresponds to the distance of j from the evader. In the case where $ts.j$ is equal to ts values of j's neighbors, j uses the d values of its neighbors to elect its parent to be the one offering the shortest distance to the evader.

Thus, the actions for j (parameterized with respect to neighbor k) in the evader-centric program is as follows.

$$\{ \text{ Evader resides at } j \ \} \longrightarrow \ p.j := j;\ ts.j := clock.j;\ d.j := 0$$
$$\square$$
$$ts.k > ts.j \ \ \vee \ \ (ts.k = ts.j \ \ \wedge \ \ d.k + 1 < d.j) \longrightarrow \ p.j := k;\ ts.j \ := \ ts.(p.j);\ d.j := d.(p.j) + 1$$

Once a tracking tree is formed, the pursuer follows this tree to reach the evader simply by querying its closest node for its parent and proceeding to the parent node. Thus, the pursuer action is as follows.

$$\{\text{Pursuer resides at } j\} \longrightarrow \quad \text{Pursuer moves to } p.j$$

9.4.1 Proof of correctness

As Figure 9.1. illustrates, if the evader is moving it may not be possible to maintain a minimum distance spanning tree.

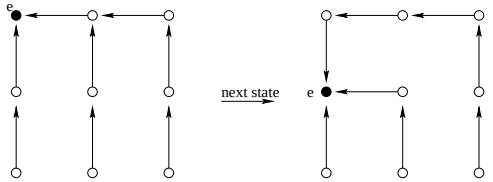

Figure 9.1. Minimum spanning tree is not maintained when evader is moving.

Note that this is a worst-case scenario and occurs when the evader speed is as fast as the communication speed of the nodes. (Tracking is not achievable if the evader is faster than the communication speed of the nodes.) In practice, node communication (25msec) is faster than the evader movement and construction of a minimum distance spanning tree is possible.

However, even in this worst case scenario, we can still prove the following properties in the absence of faults.

Theorem 1 . The tracking tree is a spanning tree rooted at the node where the evader resides and is fully constructed in at most D steps.

Proof. From the synchronized clocks assumption and the privileged detection action, $\{$Evader resides at $j\}$, it follows that the node j where the evader resides has the highest timestamp value in the network. Observe from the second node action that the $p.k$ variable at every node k embed a logical tree structure over the sensor network. Cycles cannot occur since $(\forall k : ts.k > 0 : d.(p.k) < d.k)$ [1]. Since $(\forall k : ts.k > 0 : ts.(p.k) > ts.k)$, the network is connected, and the node j where the evader resides has the highest timestamp value in the network, it follows that there exists only one tree in the network and it is rooted at j. Within at most D steps all the nodes in the network receives a message from a node that is already included in the tracking tree (due to the maximal parallelism model and the second node action), and a tracking tree covering the entire network is constructed. □

Lemma 3 . The distance between the pursuer and evader does not increase once the constructed tree includes the node where the pursuer resides.

Proof. Once the constructed tree includes the node k_x where the pursuer resides, there exists a path $k_1, k_2, \ldots, k_x$ such that $(\forall i : 1 < i \leq x : p.k_i = k_{i-1})$ and the evader resides at k_1. At any program step, if the evader moves to a neighboring node, the pursuer, being faster than the evader, also moves to the next node in the path.

Note that at each program step, any node k_i in this path may choose to change its parent, rendering a different path between the pursuer and the evader. However, observe from the second node action that, k_i changes its parent to be the neighbor that has a shorter path to the evader (higher timestamp implies shorter path since the node where evader resides has the highest timestamp and nodes execute under maximal parallelism model). Thus, the net effect is that the path length can only decrease but not increase. □

Theorem 4 . The pursuer catches the evader in at most $M + 2M * \lceil \alpha/(1-\alpha) \rceil$ steps.

Proof. Since the initial distance between the evader and the pursuer is M, after M program steps the tracking tree includes the node at which the pursuer resides. Since the evader's speed is below unit time step of the protocol execution, within this period the evader can move to at most M hops away, potentially increasing the distance between the evader and pursuer to $2M$. From Lemma 3 , it follows that this distance cannot increase in the subsequent program steps. Since the pursuer is faster than the evader, it catches the evader in at most $2M * \lceil \alpha/(1-\alpha) \rceil$ steps (follows from solving $\alpha = X/(X+2M)$ for X). □

9.4.2 Proof of stabilization

In the presence of faults variables of a node j can be arbitrarily corrupted. However, for the sake of simplicity we assume that even in the presence of faults the following two conditions hold:

[1]The predicate $(\forall i : R.i : X.i)$ may be read as "for all i that satisfy $R.i$, $X.i$ is also true".

1. always $ts.j \leq clock.j$
2. always $\{p.j \in \{nbr.j \cup \{j\} \cup \{\perp\}\}$

The first condition states that the timestamp for the detection of evader at node j is always less than the local clock at j (i.e., $ts.j$ cannot be in the future). The second condition states that the domain of $p.j$ is restricted to the set $\{nbr.j \cup \{j\} \cup \{\perp\}\}$ where $p.j = \perp$ denotes that j does not have any parent. These are both locally checkable and enforceable conditions; in order to keep the program simple we will not include the corresponding correction actions in our presentation.

Lemma 5 . The tracking tree stabilizes in at most D steps.

Proof. Since we have always $ts.j \leq clock.j$, even at an arbitrary state (which might be reached due to transient faults) the node where the evader resides has the highest timestamp value in the network. From Theorem 1 it follows that a fresh tracking tree is constructed within at most D steps and this tracking tree is a spanning tree rooted at the node where the evader currently resides. □

Theorem 6 . Starting from an arbitrarily corrupted state, the pursuer catches the evader in at most $D + 2D * \alpha/(1 - \alpha)$.

Proof. The proof follows from the proofs of Lemma 5 and Theorem 4 . □

9.4.3 Performance metrics

The evader-centric program is not energy efficient since every node communicates with its neighbor at each step of the program. That is, $\omega * N$ communications occur each step, where ω denotes the average degree of a node. The communications can be treated as broadcasts, and hence, the number of total communications per step is effectively N.

On the other hand, the tracking time and the convergence time of the evader-centric program is fast: starting from an arbitrarily corrupted state it takes at most $D+2D*\alpha/(1-\alpha)$ steps for the pursuer to catch the evader.

9.5 A PURSUER-CENTRIC PROGRAM

In this section we present a pursuer-centric solution to the pursuer-evader problem in sensor networks. Here, similar to the evader-centric program, every sensor node, j, maintains a value, $ts.j$, that denotes the latest timestamp that j knows for the detection of the evader. Initially, for all j, $ts.j = 0$. If j detects the evader, it sets $ts.j$ to its clock's value.

In this program, nodes communicate with neighbors only at the request of the pursuer: When the pursuer reaches a node j, j resets $ts.j$ to zero and directs the pursuer to a neighboring node with the highest recorded time (we use $next.j$ to denote this neighbor). Note that if all ts values of the neighbors are the same (e.g., zero), the pursuer is sent to an arbitrary neighbor. Also, if there is no pursuer at j, $next.j$ is set to $\perp$ (i.e., *undefined*).

Thus, the actions for node j in the pursuer-centric program is as follows:

$$
\begin{array}{|l|l|}
\hline
\{ \text{ Evader detected at } j \ \} \longrightarrow & ts.j := clock.j \\
\square & \\
\{ \text{ Pursuer detected at } j \ \} \longrightarrow & next.j :\in \{k \mid k \in nbr.j \ \wedge \\
 & \qquad ts.k = max(\{\ ts.l \mid l \in nbr.j\ \})\}; \\
 & ts.j := 0 \\
\hline
\end{array}
$$

The pursuer's action is as follows.

$\{$Pursuer resides at $j\}$ $\longrightarrow$	Pursuer moves to $next.j$

9.5.1 Proof of correctness

In the absence of faults, our pursuer-centric program satisfies the following properties.

Lemma 7 . If the pursuer reaches a node j where $ts.j > 0$, the pursuer catches the evader in at most $N * \alpha/(1-\alpha)$ steps.

Proof. If the pursuer reaches a node j where $ts.j > 0$, then there exists a path between the pursuer and the evader that is at most of length N. This distance does not increase in the following program steps (due to maximal parallel execution semantics and the program actions). □

In [10], it is proven that during a random walk on a graph the expected time to find N distinct vertices is $O(N^3)$. However, a recent result [19] shows that by using a local topology information (i.e., degree information of neighbor vertices) it is possible to achieve the cover time $O(N^2 logN)$ for random walk on any graph. Thus, we have:

Lemma 8 . The pursuer reaches a node j where $ts.j > 0$ within $O(N^2 logN)$ steps. □

Theorem 9 . The pursuer catches the evader within $O(N^2 logN)$ steps. □

9.5.2 Proof of stabilization

Since each node j resets $ts.j$ to zero upon a detection of the pursuer, arbitrary $ts.j$ values eventually disappear, and hence, the pursuer-centric program is self-stabilizing.

Theorem 10 . Starting from an arbitrary state, the pursuer catches the evader within $O(N^2 logN)$ steps. □

9.5.3 Performance metrics

The pursuer-centric program is energy efficient. At each step of the program only the node where the pursuer resides communicates with its neighbors. That is, ω communications occur at each step.

On the other hand, the tracking and the convergence time of the pursuer-centric program is slow: $O(N^2 logN)$ steps.

9.6 A HYBRID PURSUER-EVADER PROGRAM

In the hybrid program we combine the evader-centric and pursuer-centric approaches:

1. We modify the evader-centric program to limit the tracking tree to a bounded depth R to save energy.

2. We modify the pursuer-centric program to exploit the tracking tree structure.

We limit the depth of the tracking tree to R by means of the distance, d, variable.

$$
\begin{array}{l}
\{\text{ Evader resides at } j\ \} \longrightarrow \quad p.j := j;\ ts.j := clock.j;\ d.j := 0 \\
\ \square \\
d.k < R\ \wedge\ (ts.k > ts.j\ \ \vee\ \ (ts.k = ts.j\ \wedge\ d.k{+}1 < d.j)) \\
\qquad\qquad\qquad\qquad \longrightarrow \quad p.j := k;\ ts.j\ :=\ ts.(p.j); \\
\qquad\qquad\qquad\qquad\quad\ \ d.j := d.(p.j) + 1
\end{array}
$$

By limiting the tree to a depth R we lose the advantages of soft-state stabilization: there is no more a flow of fresh information to correct the state of the nodes that are outside the tracking tree. To achieve stabilization, we add explicit stabilization actions. Next we describe these two actions.

Starting from an arbitrarily corrupted state where the graph embedded by the parent relation on the network has cycles, each cycle is detected and removed by using the bound on the length of the path from each process to its root process in the tree. To this end, we exploit the way that we maintain the d variable: j sets $d.j$ to be $d.(p.j) + 1$ whenever $p.j \in nbr.j$ and $d.(p.j) + 1 \leq R$. The net effect of executing this action is that if a cycle exists then the $d.j$ value of each process j in the cycle gets "bumped up" repeatedly. Within at most R steps, some $d.(p.j)$ reaches R, and since the length of each path in the adjacency graph is bounded by R, the cycle is detected. To remove a cycle that it has detected, j sets $p.j$ to $\perp$ (undefined) and $d.j$ to ∞, from whereon the cycle is completely cleaned within the next R steps. Note that this action also takes care of pruning the tracking tree to height R (e.g., when the evader moves and as a result a node j with $d.j = R$ becomes $R+1$ away from the evader).

Node j also sets $p.j$ to $\perp$ (undefined) and $d.j$ to ∞ if $p.j$ is not a valid parent (e.g. $d.j \neq d.(p.j) + 1$ or $ts.j > ts.(p.j)$ or $(p.j = j\ \ \wedge\ \ d.j \neq 0)$).

We add another action to correct the fake tree roots. If a node j is spuriously corrupted to $p.j = j\ \ \wedge\ \ d.j = 0$, this is detected by explicitly asking for a proof of the evader at j.

Thus the stabilization actions for the bounded length tracking tree is as follows.

$$
\begin{array}{l}
p.j \neq \perp\ \wedge\ \ ((p.j = j\ \ \wedge\ \ d.j \neq 0)\ \ \vee\ \ ts.j > ts.(p.j) \\
\qquad\qquad \vee\ d.j \neq d.(p.j) + 1\ \ \vee\ \ d.(p.j) \geq R) \\
\qquad\qquad\qquad\qquad \longrightarrow \quad p.j := \perp;\ d.j := \infty \\
\ \square \\
p.j = j\ \ \wedge\ \ d.j = 0\ \ \wedge\ \ \neg\{\text{ Evader resides at } j\ \} \\
\qquad\qquad\qquad\qquad \longrightarrow \quad p.j := \perp;\ d.j := \infty
\end{array}
$$

We modify the node action in the pursuer-centric program only slightly so as to exploit the tracking tree structure.

$$
\begin{array}{l}
\{\text{ Pursuer detected at } j\ \} \longrightarrow \\
\qquad \text{if } (p.j \neq \perp) \text{ then } next.j := p.j \\
\qquad \text{else} \\
\qquad\quad next.j :\in \{k \mid k \in nbr.j\ \ \wedge\ \ ts.k{=}max(\{ts.l \mid l \in nbr.j\})\}; \\
\qquad\quad ts.j := 0
\end{array}
$$

Finally, the pursuer action is the same as that in Section 9.5.

9.6.1 Proof of correctness

In the absence of faults, the following lemmas and theorem follow from their counterparts in Sections 9.4 and 9.5.

Lemma 11 . The tracking tree is fully constructed in at most R steps. □

Below n denotes the number of nodes included in the tracking tree.

Lemma 12 . The pursuer reaches the tracking tree within $O((N-n)^2 log(N-n))$ steps. □

Theorem 13 . The pursuer catches the evader within $O((N-n)^2 log(N-n))$ steps. □

Since the evader is mobile, the number of nodes in the tracking tree of depth R varies over time depending on the location of the evader and the density of nodes within the R-hop neighborhood. However, the number n we use in the $O()$ notation depends only on the number of nodes included in the first tracking tree constructed. More specifically, in the $O()$ notation we use $N-n$ to denote the number of nodes that the pursuer needs to perform a random walk on to reach a node that was once involved in the tracking tree, i.e., $ts > 0$. Even though, the maximum number of nodes that the pursuer needs to visit monotonously decreases as the evader moves and new tracking trees are constructed, in our analysis we still use $N-n$, that resulted from the construction of the first tracking tree, to capture the worst case scenario.

9.6.2 Proof of stabilization

Lemma 14 . The tracking tree structure stabilizes in at most $2R$ steps.

Proof. Stabilization of the nodes within the tracking tree follows from Lemma 5 . The discussion above about the stabilization actions of the hybrid program states that a cycle outside the tracking tree is resolved within $2R$ steps. These two occur in parallel, thus system stabilization is achieved within $2R$ steps. □

Theorem 15 . Starting from an arbitrary state, the pursuer catches the evader within $O((N-n)^2 log(N-n))$ steps. □

9.6.3 Performance metrics

The hybrid program for the nodes can be tuned to be energy efficient by decreasing R since it decreases n. At each step of the program at most $n+\omega$ communications take place.

The hybrid program can also be tuned to track and converge faster by increasing R since it increases n, and the time, $O((N-n)^2 log(N-n))$ steps, a random walk takes to find the tracking tree. From that point on it takes only $R * \alpha/(1-\alpha)$ steps for the pursuer to catch the evader.

Note that there is a tradeoff between the energy-efficiency and the tracking time. In Section 9.8, we provide an example where we choose a suitable value for R to optimize both energy-efficiency and tracking time concurrently.

9.7 AN EFFICIENT VERSION OF THE HYBRID PROGRAM

In this section we present a communication- and, hence, energy-efficient version of the hybrid program. We achieve this by replacing the random walk of the pursuer with a more

energy-efficient approach, namely that of constructing a search-tree rooted at the pursuer. To this end, we first present the extended and energy-efficient version of the pursuer-centric program, and then show how this extended pursuer-centric program can be incorporated into the hybrid program.

Extended pursuer-centric program. In the extended version of the pursuer-centric program, instead of the random walk prescribed in Section 9.5, the pursuer uses *agents* to search the network for a trace of the evader. The pursuer agents idea can be implemented by constructing a (depth-first or breadth-first) tree rooted at the node where the pursuer resides. If a node j with $ts.j > 0$ is included in this *pursuer tree*, the pursuer is notified of this result along with a path to j. The pursuer then follows this path to reach j. From this point on, due to Lemma 7 , it will take at most $N * \alpha/(1-\alpha)$ steps for the pursuer to catch the evader.

This program can be seen as an extension of the original pursuer-centric program in that instead of a 1-hop tree construction (i.e., the node k where the pursuer resides contacts $nbr.k$) embedded in the original pursuer-centric program, we now employ a D-hop tree construction. To this end we change the original pursuer program as follows. The node k where the pursuer resides sets $next.j$ to $\perp$ if none of its neighbors has a timestamp value greater than 0, instead of setting $next.j$ to point to a random neighbor of j. The pursuer upon reading a $\perp$ value for the $next$ variable, starts a tree construction to search for a trace of the evader. Note that by using a depth D, the pursuer tree is guaranteed to encounter a node j with $ts.j > 0$.

Several extant self-stabilizing tree construction programs [1,11,13] suffice for constructing the pursuer tree in D steps and to complete the information feedback within another D steps. Also since the root of the pursuer tree is static (root does not change dynamically unlike the root of the tracking tree), it is possible to achieve self-stabilization of pursuer tree within D steps in an energy efficient manner. That is, in contrast to the evader-centric tracking tree program where all nodes communicate at each program step, in the pursuer tree program only the nodes propagating a (tree construction or information feedback) wave need to communicate with their immediate neighbors.

Extended hybrid program. It is straightforward to incorporate the extended version of the pursuer-centric program into the hybrid program. The only modification required is to set the depth of pursuer tree to be $D - R$ hops instead of D hops. Note that $D - R$ hops is enough for ensuring that the pursuer will encounter a trace of the evader (i.e., pursuer tree will reach a node included in the tracking tree). After a node that is/has been in the tracking tree is reached, the pursuer program in Section 9.6 applies as is.

9.7.1 Performance metrics

The extension improves the tracking and the convergence time of the pursuer-centric program from $O(N^2 logN)$ steps to $3D + N * \alpha/(1-\alpha)$ steps ($2D$ steps for the pursuer tree construction and information feedback, and D steps for the pursuer to follow the path returned by the pursuer tree program). The extended pursuer-centric program remains energy efficient; the only overhead incurred is the one-time invocation of the pursuer tree construction.

In the extended hybrid program, it takes at most $3 * (D - R)$ steps for the pursuer to reach the tracking tree. (Compare this to $O((N-n)^2 log(N-n))$ steps in the original hybrid program.) From that point on it takes $R * \alpha/(1-\alpha)$ steps for the pursuer to catch

the evader. At each step of the extended hybrid program at most n communications take place. Due to the pursuer tree computation, a one time cost of $(N - n)$ is incurred.

1-pursuer 0-evader scenario. The evader-centric program is energy efficient in a scenario where there is no evader but there is a pursuer in the system: no energy is spent since no communication is needed. On the other hand, the pursuer-centric program performs poorly in this case: at each step the pursuer queries the neighboring nodes incurring a communication cost of w. The hybrid program, since it borrows the pursuer action from the pursuer-centric program, also performs badly in this scenario.

The extended pursuer-centric program fixes this problem by modifying the pursuer tree construction to require that an answer is returned only if the evader tree is encountered. That is, if there is no evader in the network, the pursuer tree program continues to wait for the information feedback wave to be triggered, and hence, it does not waste energy.

0-pursuer 1-evader scenario. By enforcing that pursuers authenticate themselves when they join the network and notify the network when they leave, we can ensure that a tracking tree is maintained only when there is a pursuer in the system and achieve energy-efficiency.

9.8 IMPLEMENTATION AND SIMULATION RESULTS

In this section, we present implementation and simulation results, and show an example of tuning our tracking program to optimize both energy-efficiency and tracking time concurrently.

9.8.1 Implementation

We have implemented an asynchronous version of the evader-centric program on the Berkeley's Mica node platform [18] for a demonstration at the June 2002 DARPA–Network Embedded Systems Technology (NEST) retreat held in Bar Harbor, Maine.

Asynchronous program. Even though we assumed an underlying clock synchronization service for our presentation, it is possible to modify the evader-centric program slightly (only 1 line is changed) to obtain an asynchronous version. The modification is to use, at every node j, a counter variable $val.j$ that denotes the number of detections of the evader that j is aware of, instead of $ts.j$ that denotes the latest timestamp that j knows for the detection of the evader. When j detects the evader, instead of setting $ts.j$ to $clock.j$, j increases $val.j$ by one.

The extended pursuer program is also made asynchronous in a straightforward manner, since the idea of pursuer agents (a tree rooted at the pursuer) is readily implemented in the asynchronous model [1, 11, 13].

In our demonstration, a Lego Mindstorms^{TM} robot serving as a pursuer used our program to catch another Lego Mindstorms robot serving as an evader, in a 4 by 4 grid of nodes subject to a variety of faults. Figure 9.2. shows a snapshot from our demo.

The sensor nodes are embedded in a foam panel under the board. There are small rectangular holes in the board corresponding to sensor locations. A node detects an evader via its optic sensor: When the evader reaches a sensor location, its body closes the hole and triggers a "darkness" reading. The pursuer avoids this detection thanks to the glow sticks attached under its body.

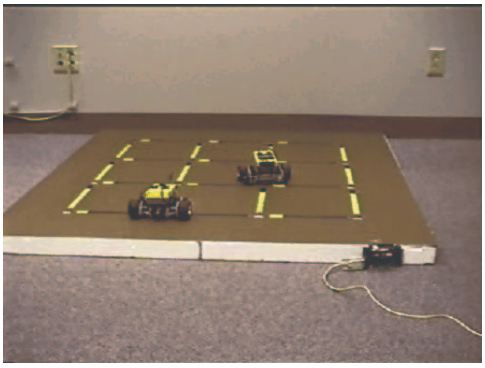

Figure 9.2. Snapshot from our demo

The colored lines on the board encode the four directions, e.g., “Long Yellow” followed by a “Short Black” indicates North. The pursuer calculates which direction it is heading after a complete line traversal.

We have soldered infrared (IR) LEDs on the nodes. Nodes blink IR LEDs at four different frequencies to communicate the four directions. On reaching a node, the pursuer detects the frequency of emitted IR signals and decides how to turn.

The evader is remotely controlled by a human playing the role of an omniscient adversary. We showed, in our demo, that despite node failures or transient corruption of the nodes, the tracking tree stabilizes to a good state, and the pursuer catches the evader by following the tree.

The pursuer robot could get disoriented on the grid or lose track of the gridlines, so we also built in stabilization in the robot program to find the gridlines from an arbitrary state. The pursuer converges to the grid from any point within the board, without falling off the edge. Upon converging, the pursuer regains its sense of direction within one complete line traversal, and starts to track the tree by following direction signals from the nodes.

Due to incorrect interaction between the motes and the pursuer robot, the robot could be driven into bad states. The stabilizing robot program was designed to tolerate such bad interactions. For example, if the pursuer reaches a failed mote, it cannot get a direction signal. The pursuer then chooses a random direction to follow. If this direction leads it off the grid, it backs up and retries till it finds a grid direction. On reaching the next non-failed node, the pursuer gets a proper direction to follow and catches the evader eventually.

We have recently ported our TinyOS sensor node code to nesC [15]; the source code for the sensors, the pursuer and evader robots (written in NQC), and video shots for the demo are available at `www.cis.ohio-state.edu/~demirbas/peDemo/`.

9.8.2 Simulation results

In the preceding sections we have presented analytical worst-case bounds on the performance of our tracking service. In this section we consider a random movement model for the evader and compare and contrast the average case performances of our tracking programs through simulation.

For our simulations, we use Prowler [24], a MATLAB based, event-driven simulator for wireless sensor network. Prowler simulates the radio transmission/propagation/reception delays of Mica2 motes [18], including collisions in ad-hoc radio networks, and the operation

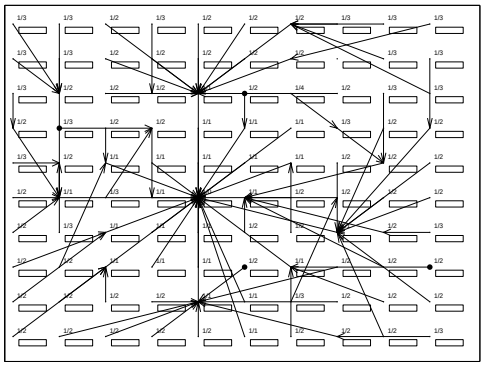

Figure 9.3. Simulation for evader-centric program

of the MAC-layer. The average transmission time for a packet is around 25 milliseconds. Our implementations are per node and are message-passing distributed programs. Our code for the simulations is also available from `www.cis.ohio-state.edu/~demirbas/peDemo`.

In our simulations, the network consists of 100 nodes arranged in a grid topology of 10-by-10. The distance between two neighboring nodes on the grid is a unit distance. The node communication radius is approximately 2 units. The evader makes a move every 2 sec, and the pursuer every 1 sec. The average move distance for both the pursuer and evader is 2 units.

Table 9.1. shows the number of total messages and the catching times for the evader-centric program, the pursuer-centric program, hybrid program with $R = 1$, and hybrid program with $R = 2$. These averages are calculated using 30 runs of these programs with random starting locations for the pursuer and the evader.

	Evader-centric	Pursuer-centric	Hybrid $R=1$	Hybrid $R=2$
Total # of mesgs	256	21	286	208
Catch time (sec)	3.3	15.7	10.7	3.9

Table 9.1. Number of messages and catching times

Figure 9.3. shows a snapshot from the simulation of the evader-centric program. The direction of the arrows denote the parent pointers at the motes. The two numbers next to a node correspond to val and d (in hops) variables at that node. The tracking tree is rooted at the evader; the evader happens to reside at the middle of the grid in that run. The tracking tree spans the entire network and may have up to 4 hops. It is costly to maintain this tree: on average a total of 256 messages are sent before the evader is captured.

Since the transmission time of a message (25msec) is much smaller than the speed of the evader, the tracking tree is effectively a minimum spanning tree. (The irregularities and long links are due to the nondeterministic nature of the radio model, fading effects, and collusions.) Hence, the pursuer catches the evader within about 3-4 moves: average catching time is 3.3 sec.

Figure 9.4. shows a snapshot from the pursuer-centric program. In this run, the evader started in the middle of the network and moved to the upper-left corner within 16 moves. The pursuer started at the lower-left corner, randomly wandered around for a while, found

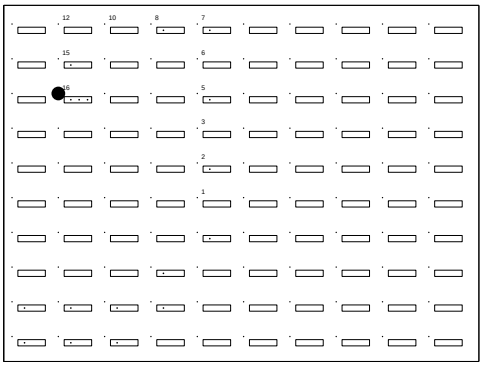

Figure 9.4. Simulation for pursuer-centric program

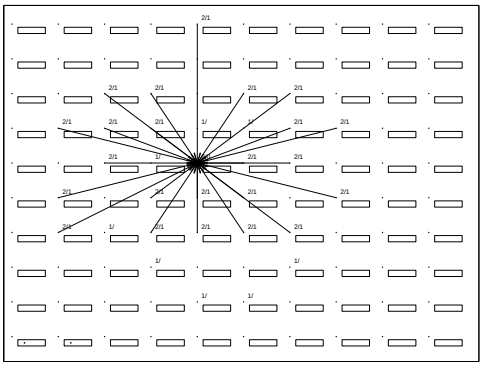

Figure 9.5. Simulation for hybrid program for R=1

a node that was visited by the evader, and followed these tracks to catch the evader at the upper-left corner (a dot in the rectangle denotes that pursuer has visited the corresponding node).

Since the pursuer-centric program does not maintain a tracking tree, the total number of messages sent is low (21). On the other hand, it takes more time for the pursuer to find a track of the evader, hence catching time is high (15.7 sec).

Figure 9.5. shows a snapshot from the hybrid program with $R = 1$ (hybrid1), and Figure 9.6. shows the hybrid program with $R = 2$ (hybrid2). Our simulations show that hybrid2 performs better than hybrid1: Both the total number of messages sent and the catching time of hybrid2 is significantly smaller than those of hybrid1.

Hybrid1 cannot provide a good coverage over the network with its one-hop tracking tree. Hence the pursuer wanders around for 5-10 moves before it can encounter a node that had some info about the evader (a node that is/has been part of the tracking tree). Due to this wandering around time, the catching time increases and energy is wasted for maintaining a tracking-tree for an elongated time. (Note that the maintenance of a 1-hop tree is not achievable only by a broadcast of the root node. The leaf nodes also broadcasts messages; these broadcasts are required for informing the nodes that are to be pruned, i.e., the nodes that were included in the previous tracking tree, but that are outside the new tracking tree as a result of evader movement and accompanying root node change.)

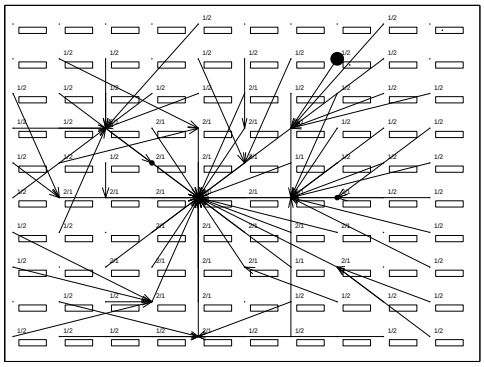

Figure 9.6. Simulation for hybrid program for R=2

Hybrid2, on the other hand, provides a reasonable coverage over the network, hence the pursuer discovers the track of the evader earlier than that of hybrid1. The tracking tree is maintained to $R = 2$, and thus, after each evader move hybrid2 sends more messages than hybrid1. But since the catching time is significantly shortened in hybrid2, the 2-hop tracking tree is maintained only for this short time. As a result, the total number of messages sent by hybrid2 is less than that of hybrid1.

Hybrid2 optimizes both energy-efficiency and tracking time concurrently. The total number of messages sent by hybrid2 is less than that of the evader-centric program (208 versus 256), and the catching time of hybrid2 is comparable to that of the evader-centric program (3.9 sec versus 3.3 sec). (Hybrid program with $R = 3$ gives similar results to the evader-centric program, and is omitted from our discussion.)

9.9 DISCUSSION AND RELATED WORK

In this paper we have investigated a pursuer-evader game for sensor networks. More specifically, we have presented a hybrid, tunable, and self-stabilizing program to solve this problem. We proved that the pursuer catches the evader even in the presence of faults.

For the sake of simplicity, we have adopted a shared-memory model in our presentation; our results are still valid for message passing memory model. We have provided message-passing implementations of our programs in Section 9.8. Note that the semantics of the message-passing program is event-based execution (e.g., upon receiving a message or detecting an evader/pursuer), rather than maximal parallelism.

Energy efficiency. We have demonstrated that our program is tunable for tracking speed or energy efficiency. Our program is also tunable for stabilization speed or energy efficiency. The periodicity of soft-state updates for stabilization should be kept low if the faults are relatively rare in the network. For example, in the absence of faults, the first action (i.e., $\{$Evader resides at $j\}$ action) need not be executed unless the evader moves to a different node. Similarly, the stabilization actions (actions 3 and 4 of the hybrid program) can be executed with low frequency to conserve energy.

Another way to improve the energy-efficiency is to maintain the tracking tree over a small number of nodes. For example, hierarchical structuring can be employed to maintain tracking information with accuracy proportional to the distance from the evader. Also

maintaining the tracking tree in a directional manner and only up to the location of the pursuer will help conserve energy.

Related work. Several self-stabilizing programs exist for tree construction ([1, 11, 13] to name a few). However, our evader-centric program is unique in the sense that a spanning tree is maintained even though the root changes dynamically.

A self-stabilizing distributed directory protocol based on path reversal on a network-wide, fixed spanning tree is presented in [17]. The spanning tree is initialized to guarantee a reachability condition: following the links from any node leads to the evader. When the evader moves from a node j to another node k, all the links along the path from j to k in the spanning tree are reversed. This way, the tree always guarantees the reachability condition. This protocol suffers from a nonlocal update problem because it is possible to find at least two adjacent nodes j, k in the network such that the distance between j and k in the overlayed spanning tree structure is twice the height of the tree (i.e., equal to the diameter of the network). An evader that is dithering between these two nodes may cause the protocol to perform nonlocal updates for each small move, and would result in a scenario where the pursuer is never able to catch the evader. In contrast, our protocol maintains a dynamic tree and does not suffer from the nonlocal update problem.

In our program, we choose to update the location of the evader immediately. In [9], three strategies for when to update the location the evader (time-based, number of movements-based, and distance-based) are evaluated with respect to their energy efficiency.

Relating to the idea of achieving energy efficiency by using a small number of nodes, Awerbuch and Peleg [6] present a local scheme that maintains tracking information with accuracy proportional to the distance from the evader. They achieve this goal by maintaining a hierarchy of $logD$ regional directories (using the graph-theoretic concept of *regional matching*) where the purpose of the i'th level regional directory is to enable a pursuer to track the evader residing within 2^i distance from it. They show that the communication overhead of their program is within a polylogarithmic factor of the lower bound. Loosely speaking, their regional matching idea is an efficient realization of our pursuer-centric program and their forwarding pointer structure is analogous to our tracking tree structure.

By way of contrast, their focus is on optimizing the complexity during the initialized case, whereas we focus on optimizing complexity during stabilization as well. That is, we are interested in (a) tracking that occurs while initialization is occuring; in other words, soon after the evader joins the system, and (b) tracking that occurs from inconsistent states; in other words, if the evader moves in an undetectable/unannounced manner for some period of time yielding inconsistent tracks. Their complexity of initialization is $O(E\ log^4 N)$ where E is the number of edges in the graph and N is the number of nodes. Thus, brute force stabilization of their structure completes in $O(E\ log^4 N)$ time as compared with the $2R$ steps it takes in our extended hybrid program.

We have recently found that [14] if we restrict the problem domain to tracking in planar graphs, it is possible to optimize the tracking time in the presence of faults as well as the communication cost and tracking time in the absence of faults. A topology change triggers a global initialization in Awerbuch and Peleg's program since their m-regional matching structure depends on a non-local algorithm that constructs sparse covers [5]. Assuming that the graph is planar (neither [6] nor this paper assumes planarity), a local and self-stabilizing clustering algorithm [21] for constructing the m-regional matching structure is achievable, and hence, it is possible to deal with topology changes locally.

The concept of self-stabilization is particularly useful for dealing with unanticipated and undetectable faults [3]. To achive such an ambitious goal, self-stabilization assumes

for convenience that no further faults occur within the stabilization period. It is possible to improve stabilizing programs by adding masking fault-tolerance for common and detectable faults; this way occurence of trivial, common faults during stabilization can be masked immediately, and does not affect the stabilization time. The design of this type of fault-tolerance, known as multi-tolerance, is discussed in [2].

Moreover, for the type of faults for which masking is impossible or infeasible, preventing them from spreading is useful for achieving scalability of stabilization for large-scale networks. To this end, several fault-containment techniques [4, 7, 16, 22]. have been proposed in the self-stabilization literature.

Furthermore, by choosing a weaker invariant it is possible to show that the stabilization of our tracking programs are unaffected by common faults such as message losses or node fail-stops. That is, by accepting a degraded tracking performance in the presence of these faults, we can show that stabilization to a weaker invariant —e.g., a tracking tree, albeit not the optimal tree— is still achievable under message losses and node fail-stops.

Future work. We have found several variations of the pursuer-evader problem to be worthy of study, where we change for instance the communication time between nodes, the numbers of pursuers and evaders, and the range of a move. Especially of interest to us are general forms of the tracking problem where efficient solutions can be devised by hybrid control involving traditional control theory and self-stabilizing distributed data structures (such as tracking trees and regional directories).

REFERENCES

1. A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.
2. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.
3. A. Arora and Y-M Wang. Practical self-stabilization for tolerating unanticipated faults in networked systems. Technical Report OSU-CISRC-1/03-TR01, The Ohio State University, 2003.
4. A. Arora and H. Zhang. LSRP: Local stabilization in shortest path routing. In *IEEE-IFIP DSN*, pages 139–148, June 2003.
5. B. Awerbuch and D. Peleg. Sparse partitions (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.
6. B. Awerbuch and D. Peleg. Online tracking of mobile user. *Journal of the Association for Computing Machinery*, 42:1021–1058, 1995.
7. Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *ACM PODC*, pages 33–42, 2003.
8. A. Bar-Noy and I. Kessler. Tracking mobile users in wireless communication networks. In *INFOCOM*, pages 1232–1239, 1993.
9. A. Bar-Noy, I. Kessler, and M. Sidi. Mobile users: To update or not to update? In *INFOCOM*, pages 570–576, 1994.
10. G. Barnes and U. Feige. Short random walks on graphs. *SIAM Journal on Discrete Mathematics*, 9(1):19–28, 1996.
11. N.S. Chen and S.T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters (IPL)*, 39:147–151, 1991.
12. Y. Choi, M. Gouda, M. C. Kim, and A. Arora. The mote connectivity protocol. *Proceedings of the International Conference on Computer Communication and Net- works (ICCCN-03)*, 2003.

13. A. Cournier, A.K. Datta, F. Petit, and V. Villain. Self-stabilizing PIF algorithms in arbitrary networks. *International Conference on Distributed Computing Systems (ICDCS)*, pages 91–98, 2001.

14. M. Demirbas, A. Arora, T. Nolte, and N. Lynch. STALK: A self-stabilizing hierarchical tracking service for sensor networks. Technical Report OSU-CISRC-4/03-TR19, The Ohio State University, April 2003.

15. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NESC language: A holistic approach to network embedded systems. Submitted to the ACM SIGPLAN(PLDI), June 2003.

16. S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *ACM PODC*, pages 45–54, 1996.

17. M.P. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th International Symposium on Distributed Computing*, pages 209–219, oct 2001.

18. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, pages 93–104, 2000.

19. S. Ikeda, I. Kubo, N. Okumoto, and M. Yamashita. Local topological information and cover time. Research manuscript, 2002.

20. M. Jayaram and G. Varghese. Crash failures can drive protocols to arbitrary states. *ACM Symposium on Principles of Distributed Computing*, 1996.

21. V. Mittal, M. Demirbas, and A. Arora. LOCI: Local clustering in large scale wireless networks. Technical Report OSU-CISRC-2/03-TR07, The Ohio State University, February 2003.

22. M. Nesterenko and A. Arora. Local tolerance to unbounded byzantine faults. In *IEEE SRDS*, pages 22–31, 2002.

23. E. Pitoura and G. Samaras. Locating objects in mobile computing. *Knowledge and Data Engineering*, 13(4):571–592, 2001.

24. G. Simon, P. Volgyesi, M. Maroti, and A. Ledeczi. Simulation-based optimization of communication protocols for large-scale wireless sensor networks. *IEEE Aerospace Conference*, March 2003.

25. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.

26. A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27, 2003.