A Pursuer-Evader Game for Sensor Networks

Murat Demirbas[†] *Anish Arora*[†] *Mohamed Gouda*[‡]

† Department of
Computer & Info. Science
The Ohio State University
Columbus, OH 43210 USA

‡ Department of
Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA

Abstract

In this paper we present a self-stabilizing program for solving a pursuer-evader problem in sensor networks. The program can be tuned for tracking speed or energy efficiency. In the program, sensor motes 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.

Keywords : Sensor networks, pursuer-evader games, distributed algorithms, self-stabilization, tracking speed, energy efficiency.

---

[0]This work was partially sponsored by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and two grants from Microsoft Research.

Contact information. Email: demirbas@cis.ohio-state.edu, Tel: +1 614 688 4637, Fax: +1 614 292 2911.

1 Introduction

Due to its importance in military contexts, pursuer-evader tracking has received significant attention [3, 4, 15, 16] 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 micro-electromechanical 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 [13]. 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). Note that 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 6, 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 3 and a pursuer-centric program in Section 4. In Section 5, we present the tunable, hybrid program combining the previous two programs. We present an efficient version of the hybrid program in Section 6. Finally, we discuss related work and make concluding remarks in Section 7.

2 The Problem

System model. A sensor network consists of a (potentially large) number of sensor nodes (called motes). Each mote is capable of receiving/transmitting messages within its field of communication. All motes within this communication field are its neighbors; we denote this set for mote 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 [8].)

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

Both the pursuer and the evader are mobile: each can atomically move from one mote 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 mote at which it resides.

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

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

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

Each action has the form:

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

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 mote actions. At each state, each mote executes all actions that are enabled in that state. (Execution of multiple enabled actions in a mote 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 mote 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 mote has a clock, that is synchronized with the clocks of other motes. (Real time advances with each program step, in a non-Zeno sense.) This assumption is reasonably implemented at least for Mica motes [11]. Later, in Section 7, 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 motes. We use $var.j$ to denote the variable var residing at j. We use $[]$ 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 mote. 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 mote j parameterized over k abbreviates the following set of actions:

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

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 mote 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 motes in the sensor network, D the diameter of the network, and M the distance between the pursuer and evader. 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.

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

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 mote, the mote immediately detects this fact (i.e., the detection actions have priority over normal mote actions and are fired instantaneously).

Fault model. Transient faults may corrupt the program state. Transient faults may also fail-stop or restart motes (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 P eventually recovers to a state from where its specification is satisfied.

3 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 mote, 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 mote 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 mote that j received the maximum timestamp value. 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 mote j to denote 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 the neighbors to elect its parent to be the one offering the shortest distance to the evader. Thus, the actions for mote j (parameterized with respect to neighbor k) in the evader-centric program is as follows.

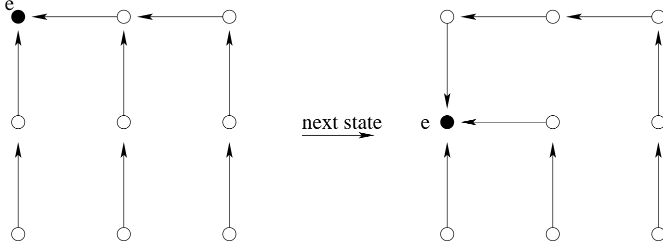
$$\begin{array}{l} \{\text{ Evader resides at } j \text{ }\} \\ \quad \longrightarrow \quad p.j := j;\ ts.j := clock.j;\ d.j := 0 \\ \ [] \\ ts.k > ts.j \ \vee\ (ts.k = ts.j \ \wedge\ d.k + 1 < d.j) \\ \quad \longrightarrow \quad p.j := k;\ ts.j := ts.(p.j); \\ \qquad\qquad\ \ d.j := d.(p.j) + 1 \end{array}$$

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

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

3.1 Proof of correctness

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



However, we can still prove the following theorem.

Theorem 1. The tracking tree is a spanning tree rooted at the mote where the evader resides.

Proof. From the synchronized clocks assumption and the privileged detection action, {Evader resides at j}, it follows that the mote j where the evader resides has the highest timestamp value in the network. Observe from the second mote action that the $p.k$ variable at every mote 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 mote 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. □

Corollary 2. The tracking tree is fully constructed in at most D steps.

Proof. Within at most D steps all the motes in the network receives a message from a mote that is already included in the tracking tree (due to the maximal parallelism model and the second mote action). From Theorem 1 it follows that 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 mote where the pursuer resides.

[1] A formula $(op\ i : R.i : X.i)$ denotes the value obtained by performing the (commutative and associative) *op* on the $X.i$ values for all i that satisfy $R.i$. Where $R.i$ is true, we omit $R.i$. As a special case, where *op* is conjunction, we write $(\forall i : R.i : X.i)$ which may be read as "if $R.i$ is true then so is $X.i$".

Proof. Once the constructed tree includes the mote 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 each program step, any mote 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 mote action that, k_i changes its parent only if the new parent has a shorter path to the evader (higher timestamp implies shorter path since the mote where evader resides has the highest timestamp and motes execute under maximal parallelism model).

At any program step, if the evader moves to a neighboring mote, the pursuer, being faster than the evader, also moves to the next mote in the path. 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 mote at which the pursuer resides. 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). □

3.2 Proof of stabilization

In the presence of faults variables of a mote 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. 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 mote 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 mote where the evader resides has the highest timestamp value in the network. From Corollary 2 and 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 mote 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. □

3.3 Performance metrics

The evader-centric program is not energy efficient since every mote 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 mote. 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.

4 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 mote, 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, motes communicate with neighbors only at the request of the pursuer: When the pursuer reaches a mote j, j resets $ts.j$ to zero and directs the pursuer to a neighboring mote 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 mote j in the pursuer-centric program is as follows:

```
{ Evader detected at j }
   ⟶      ts.j := clock.j
[]
{ Pursuer detected at j }
   ⟶      next.j :∈ {k | k ∈ nbr.j ∧
                 ts.k = max({ ts.l | l ∈ nbr.j })};
           ts.j := 0
```

The pursuer's action is as follows.

```
{Pursuer resides at j}
               ⟶      Pursuer moves to next.j
```

4.1 Proof of correctness

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

Proof. If the pursuer reaches a mote 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 [6], 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 [12] 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 mote j where $ts.j > 0$ within $O(N^2 logN)$ steps. □

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

4.2 Proof of stabilization

Since each mote 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. □

4.3 Performance metrics

The pursuer-centric program is energy efficient. At each step of the program only the mote 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.

5 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.

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

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 motes that are outside the tracking tree. To achieve stabilization, we add explicit stabilization actions. Next we describe these two actions.

For the case where the initial graph 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 mote j with $d.j = R$ becomes $R + 1$ away from the evader).

Mote 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 mote 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.

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

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

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

Finally, the pursuer action is the same as that in Section 4.

5.1 Proof of correctness

The following lemmas and theorem follow from their counterparts in Sections 3 and 4.

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

Below n denotes the number of motes 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. □

5.2 Proof of stabilization

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

Proof. Follows from Lemma 5 and the discussion above about the stabilization actions of the hybrid program. □

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

5.3 Performance metrics

The hybrid program for the motes can be tuned to be energy efficient. 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. The random walk now takes $O((N-n)^2 log(N-n))$ steps to find the tracking tree. From that point on it takes $R*\alpha/(1-\alpha)$ steps for the pursuer to catch the evader.

6 An efficient version of the hybrid program

In this section we present an efficient version of the hybrid program. To this end, we first present an extended 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 4, 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 bread-first) tree rooted at the mote where the pursuer resides. If a mote 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 mote 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 mote 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 mote j with $ts.j > 0$.

Several extant self-stabilizing tree construction programs $[1, 7, 9]$ 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 motes communicate at each program step, in the pursuer tree program only the motes 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 mote included in the tracking tree).

6.1 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, however, due to the pursuer tree computation, a one time cost of $2 * (D - R)$ 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 motes 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, a similar improvement in the energy-efficiency is obtained.

7 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; our results are still valid for message passing memory model. Note that the semantics of the message-passing program would be event-based execution (e.g., upon receiving a message or detecting the evader), rather than maximal parallelism.

Asynchronous program. Even though we assumed an underlying clock synchronization service, it is possible to modify the evader-centric program slightly to obtain an asynchronous version. The modification is to use, at every mote 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.

Implementation. We have implemented the asynchronous version of the evader-centric program on the Berkeley's Mica mote platform [11] for a demonstration at the June 2002 DARPA–NEST retreat held in Bar Harbor, Maine. 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 motes subject to a variety of faults. We have recently ported the code to

nesC [10]; the source code is available at `www.cis.ohio-state.edu/~demirbas/peDemo`.

Energy efficiency. We have demonstrated that our program is tunable for tracking speed or energy efficiency. Our program is also be 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 mote. 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 motes. 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, 7, 9] 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.

In our program, we choose to update the location of the evader immediately. In [5], 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 [3] present a local scheme that maintains tracking information with accuracy proportional to the distance from the evader[2]. 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 [2]. Assuming that the graph is planar (neither [3] nor this paper assumes planarity), we present in [14] a local and self-stabilizing clustering algorithm for constructing the m-regional matching structure, and hence, we are able to deal with topology changes locally.

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 motes, the communication model to be message broadcast instead of shared memory, the numbers of pursuers and evaders, the range of a move, and the semantics of computation to be interleaving instead of maximal parallelism.

Especially of interest to us are general forms of the tracking problem where efficient solutions can be de-

[2] We thank Nancy Lynch for bringing this paper to our attention.

vised 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] B. Awerbuch and D. Peleg. Sparse partitions (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[3] B. Awerbuch and D. Peleg. Online tracking of mobile user. *Journal of the Association for Computing Machinery*, 42:1021–1058, 1995.

[4] A. Bar-Noy and I. Kessler. Tracking mobile users in wireless communication networks. In *INFOCOM*, pages 1232–1239, 1993.

[5] A. Bar-Noy, I. Kessler, and M. Sidi. Mobile users: To update or not to update? In *INFOCOM*, pages 570–576, 1994.

[6] G. Barnes and U. Feige. Short random walks on graphs. *SIAM Journal on Discrete Mathematics*, 9(1):19–28, 1996.

[7] N.S. Chen and S.T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters (IPL)*, 39:147–151, 1991.

[8] Y. Choi, M. Gouda, M. C. Kim, and A. Arora. The mote connectivity protocol. Technical Report TR03-08, Department of Computer Sciences, The University of Texas at Austin, 2003.

[9] 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.

[10] 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.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, 2000.

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

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

[14] 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. Submitted to PODC'03.

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

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