Trail: A Distance Sensitive WSN Service For Distributed Object Tracking

Abstract. Distributed observation and control of mobile objects via static wireless sensors demands timely information in a *distance sensitive* manner: information about closer objects is required more often and more quickly than that of farther objects. In this paper, we present a wireless sensor network protocol, *Trail*, that supports distance sensitive tracking of mobile object by in-network subscribers upon demand. *Trail* achieves a *find* time that is linear in the distance from the subscriber to the object, via a distributed data structure that is updated only locally when objects move. *Trail* seeks to minimize the size of the data structure. Moreover, *Trail* is reliable, fault-tolerant and energy-efficient, despite the network dynamics that are typical of wireless sensor networks. We evaluate the performance of *Trail* by simulations in a 90-by-90 sensor network and report on 105 node experiments in the context of a pursuer-evader control application.

1 Introduction

Tracking of mobile objects has received significant attention in the context of cellular telephony, mobile computing, and military applications [1–4]. In this paper, we focus on the tracking of mobile objects using a network of static wireless sensors. Examples of such applications include those that monitor objects [5–7], as well as applications that "close the loop" by performing tracking-based control; an example is a pursuer-evader tracking application, where a controller's objective is to minimize the catch time of evaders.

We are particularly interested in large scale WSN deployments. Large networks motivate several tracking requirements. First, queries for locations of objects in a large network should not be answered from central locations as the querier may be close to the object but still have to communicate all the way to a central location. Such a solution not only increases the latency but also depletes the intermediate nodes of their energy. Plus, answering queries locally may also be important for preserving the correctness of applications deployed in large WSNs. As a specific example, consider an intruder-interceptor application where a large number of sensor nodes lie along the perimeter that surrounds a valuable asset. Intruders enter the perimeter with the intention of crossing over to the asset and the objective of the interceptors is to "catch" the intruders as far from the asset as possible. In this case, it has been shown [8] that there exist Nash equilibrium conditions which imply that, for satisfying optimality constraints, the latency with which an interceptor requires information about the intruder it is tracking depends on the relative locations of the two: the closer the distance, the smaller the latency. This requirement is formalized by the property of *distance sensitivity* for querying, i.e, the cost in terms of latency and number of messages for returning the location of a mobile object grows linearly in terms of the distance between the object and the querier.

Second, tracking services for large networks must eschew solutions with disproportionate update costs that update object locations across the network even when the object moves only by a small distance. This requirement is formalized by the property of *distance sensitivity* for updates, i.e., the cost of an update is proportional to the distance moved by the object.

Third, for large networks it is critical that object locations or pointers to the objects be maintained across only a minimum set of nodes across the network. Longer tracks have a higher cost of initialization and given that network nodes may fail due to energy depletion or hardware faults, longer tracks increase the probability of a failed node along a track as well as increase the cost of detecting and correcting failures in the track. This requirement motivates us to eschew solutions that hierarchically partition the network into a fixed number of levels [1, 3, 4] Hierarchical partitions not only yield longer track lengths, these solutions also tend to be sensitive to the failures of nodes higher up in the hierarchy.

Finally, even though solutions should designed to accommodate large networks, they should also be simple, energy-efficient and robust for use in small or medium networks.

Contributions: In this paper, we use geometric ideas to design an energy-efficient, fault-tolerant and hierarchy-free WSN service, *Trail*, that supports tracking-based WSN applications. The specification of *Trail* is to return the location of a particular object in response to an in-network subscriber issuing a *find* query regarding that object. To this end, *Trail* maintains a tracking data structure by propagating mobile object information only locally, and satisfying the distance sensitivity requirement. *Trail* avoids the need for hierarchies by determining anchors for the tracking paths on-the-fly based on the motion of objects; this allows for minimizing the length of tracking paths. *Trail* maintains tracks from each object to only one well-known point, namely, the center of the network; these tracks are *almost* straight to the center, with a stretch factor close to 1. We analytically compare the performance of *Trail* with that of other hierarchy based solutions for tracking objects and as seen in Fig. 11 in Section 7, *Trail* is more efficient than other solutions. *Trail* has about 7 times lower updates costs at almost equal $find$ costs. By using a tighter tracking structure, we are also able to decrease the upper bound *find* costs at larger distances and thereby decrease the average find cost across the network. By not relying on hierarchies *Trail* can tolerate faults more locally as well.

Trail is a family of protocols. Refinements of a basic *Trail* protocol are well suited for different network sizes and *find*/*update* frequency settings: One refinement is to tighten its tracks by progressively increasing the rate at which the tracking structure is updated; while this results in updating a large part of the tracking structure per unit move, which is for large networks still *update* distance sensitive, it significantly lowers the *find* costs for objects at larger distances. Another refinement increases the number of points along a track, i.e, progressively loosens the tracking structure in order to decrease the *find* costs and be more $find-centric$ when object *updates* are less frequent or objects are static; as an extreme case, the *find* can simply follow a straight line to the center. Moreover, *Trail* increasingly centralizes *update* and $find$ as the network size decreases.

Organization of the paper: In Section 2, we describe the system model and problem specification. In Section 3, we design a basic *Trail* for a 2-d real plane. Then, in Section 4, we present an implementation of the basic *Trail* protocol for a 2-d sensor network grid. In Section 5, we discuss refinements of the basic *Trail* protocol. In Section 6, we present results of our performance evaluation. In Section

7, we discuss related work and, in Section 8, we make concluding remarks and discuss future work.

2 Model and Specification

The system consists of a set of *mobile objects*, and a network of static *nodes* that each consist of a sensing component and a radio component. Tracking applications execute on the mobile objects and use the sensor network to track desired mobile objects. Object detection and association services execute on the nodes, as does the desired *Trail* network tracking service. The object detection service is an orthogonal service to object tracking, and as such we consider it separately. The object detection service assigns a unique id, P, to every object detected by nodes in the network and stores the state of P at the node j that is closest to the object P. This node is called the *agent* for P and can be regarded as the node where P resides. The association service can be implemented in a centralized [9] or distributed [10] fashion; the latter approach would suit integration with *Trail*.

Trail Network Service: *Trail* maintains an in-network tracking structure, $trail_P$, for every object P. *Trail* supports two functions: *find(P, Q)*, that returns the state of the object P, including its location at the current location of the object Q issuing the query and *move(P, p', p)* that updates the tracking structure when object P moves from location p' to location p.

Definition 1 (*find*(P, Q) Cost). *The cost of the* findP(P, Q) function is the total communication cost of reaching the current location of P starting from the current location of Q.

Definition 2 (*move(P, p', p)* Cost). *The cost of the move(P, p', p) function is the total communication cost of updating $trail_P$ to the new location p and deleting the tracking structure to the old location p'.*

To simplify our presentation, we first describe *Trail* in a 2-d real plane. We then refine the *Trail* protocol to suitably implement in a dense connected grid model of a WSN. We describe this model in Section 4.

3 Trail

In this section, we use geometric ideas to design *Trail* for a bounded 2-d real plane. Let C denote the center of this bounded plane.

3.1 Tracking Data Structure

We maintain a tracking data structure for each object in the network. Let P be an object being tracked, and p denote its location on the plane. Let d_{pC} be the distance of p from the center C. We denote the tracking data structure for object P as $trail_P$. Before we formally define this tracking structure, we give a brief overview.

Overview: If $trail_P$ is defined as a straight line from C to P, then every time the object moves, $trail_P$ has to be updated starting from C. This would not be a distance sensitive approach. Hence we form $trail_P$ as a set of *trail segments* and update only a portion of the structure depending upon the distance moved. The number of *trail segments* in $trail_P$ increases as d_{pC} increases. Note that we do not partition the network into a hierarchy and assign roles to specific nodes in the network. Rather, the end points of the *trail segments* serve as *anchors* to update the

tracking structure when an object moves. The anchor point from where the update is started depends on the distance moved. Only, when P moves a sufficiently large distance, $trail_P$ is updated all the way from C. We now formally define $trail_P$.

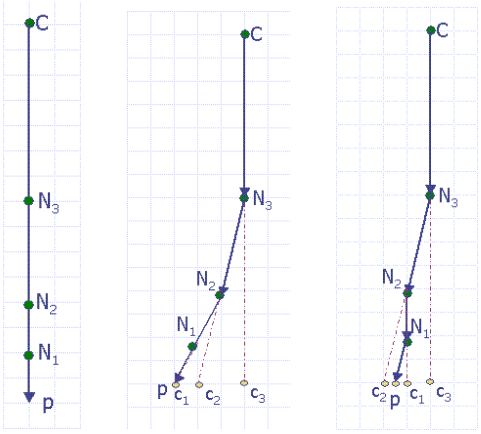
Definition 3 ($trail_P$)**.** *The tracking data structure for object P,*$trail_P$*, for* $d_{pC} \geq 1$ *is a path obtained by connecting any sequence of points* $(C, N_{max}, ..., N_k, ..., N_1, p)$ *by line segments, where* $max \geq 1$*, and there exist auxiliary points* $c_1..c_{max}$ *that satisfy the properties* $(P1)$ *to* $(P4)$ *below.*

For brevity, let N_k *be the level* k *vertex in* $trail_P$*; let the level* k trail segment *in* $trail_P$ *be the segment between* N_k *and* N_{k-1} *; let* $Seg(x, y)$ *be any line segment between points* x *and* y *in the network.*

- *(P1):* $dist(c_k, N_k) = 2^k$*, (*$max \geq k \geq 1$*).*
- *(P2):* N_{k-1}*, (*$max \geq k \geq 1$*), lies on* $Seg(N_k, c_{k-1})$*;* N_{max} *lies on* $Seg(C, c_{max})$*.*
- *(P3):* $dist(p, c_k) < 2^{k-b}$*, (*$max \geq k \geq 1$*) and* $b \geq 1$ *is a constant.*
- *(P4):* $max = \lceil (log_2(dist(C, c_{max}))) \rceil - 1$*.*

If $(d_{pC} = 0)$*,* $trail_P$ *is* C*; and if* $(0 \leq d_{pC} < 1)$*,* $trail_P$ *is* $Seg(C, p)$*.* □

Observations about $trail_P$**:** From the definition of $trail_P$, we note that the auxiliary points $c_1..c_{max}$ are used to mark vertices $N_1..N_{max}$ of $trail_P$. $P1$ and $P2$ describe the relation between the auxilliary points and the vertices of $trail_P$. Given $trail_P$, points $c_1..c_{max}$ are uniquely determined using $P1$ and $P2$. Similarly given p and $c_1, ..c_{max}$, $trail_P$ is uniquely determined. By property $P3$, the maximum separation between p and any auxiliary point c_k decreases exponentially as k decreases from max to 1. Note that we do not partition the network into a fixed number of levels. Rather, the value of max which denotes the number of *trail segments* in $trail_P$, depends on the distance of P from C.



(a) Eg.1 (b) Eg.2 (c) Eg.3

Fig. 1. Examples of Trail to an Object P

We now show 3 examples of the tracking structure. In this figure, $b = 1$. Fig. 1(a) shows $trail_P$ when $c_3, ..c_1$ are collocated. When P moves away from this location, $trail_P$ is updated and Fig. 1(b) shows an example of $trail_P$ where $c_2, ..c_1$ are displaced from c_3. In Fig. 1(b), $dist(p, c_2) = 2$ units, $dist(p, c_1) = 1$ unit, $dist(p, c_1) < 1$

units. Moreover, N_3 lies on $Seg(C, c_3)$, N_2 lies on $Seg(N_3, c_2)$ and so on. In Fig. 1(c) we show an example of a zig zag trail to an object P, when P moves away from c_3 and then moves back in the opposite direction.

3.2 Updating the trail

We now describe a procedure to update the tracking structure when object P moves from location p' to p such that the properties of the tracking structure are maintained and the cost of update is distance sensitive.

Overview: When an object moves distance d away, if the distance $dist(c_1, p)$ is less than 1, then the trail is updated by replacing $segment(N_1, p')$ with $segment(N_1, p)$. Otherwise, we find the minimal index m, along $trail_P$ such that $dist(p, c_j) < 2^{j-b}$ for all j such that $max \geq j \geq m$ and $trail_P$ is updated starting from N_m. In order to update $trail_P$ starting from N_m, we find new vertices $N_{m-1}...N_1$ and a new set of auxiliary points $c_{m-1}...c_1$. Let $N'_{m-1}...N'_0$ and $c'_{m-1}...c'_1$ denote the old vertices and old auxiliary points respectively. Starting from N_m, we follow a recursive procedure to update $trail_P$. This procedure is stated below:

Update **Algorithm:**

1. If $dist(p, c_1) \geq 1$, then let m be the minimal index on the trail such that $dist(p, c_j) < 2^{j-b}$ for all j such that $max \geq j \geq m$.
2. $k = m$
3. while $k > 1$
   (a) $c_{k-1} = p$; Now obtain N_{k-1} using property $P2$ as follows: the point on segment N_k, c_{k-1}, that is 2^{k-1} away from c_{k-1}.
   (b) $k = k - 1$

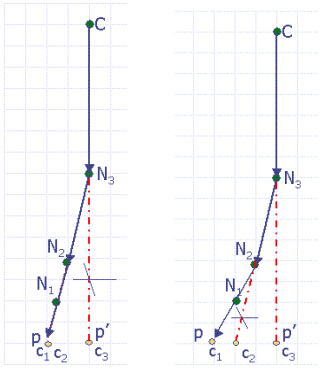
If no indices exist such that $dist(c_k, p) < 2^{k-1}$, then the trail is created starting from C. This could happen if the object is new or if the object has moved a sufficiently large distance from its original position. In this case, max is set to $(\lceil log_2(d_p) \rceil) - 1$. c_{max} is set to p. N_{max} is marked on $segment(C, p)$ at distance 2^{max} from c_{max}. Step 1 is executed with $k = max$. □

Fig. 2 illustrates an update operation, when $b = 1$. In Fig. 2(a), $dist(p, p')$ is 2 units. Hence update starts at N_3. Initially c_3, c'_2, c'_1 are at p'. We use the *update* algorithm to determine new c_2, c_1 and thereby the new N_2, N_1. Using step $(3a)$ of the update algorithm, the new c_2 and c_1 lie at p. The vertex N_2 then lies on $segment(N_3, c_2)$ and N_1 lies on $segment(N_2, c_1)$. In Fig. 2(b), P moves further one unit. Hence update now starts at N_2. Using step $(3a)$ of the update algorithm, the new c_1 lies at p and N_1 lies on $segment(N_2, c_1)$.

Lemma 1. *The update algorithm for* Trail *yields a path that satisfies* $trail_P$.

Proof. 1. Let m be the index at which *update* starts. By the condition in step 1, $dist(c_j, p) < 2^{j-b}$ for all $max \geq j \geq m$. Now, for $m > j \geq 1$, $c_j = p$. Therefore for $m > j \geq 1$, $dist(c_j, p) < 2^{j-b}$. Thus property $P3$ is satisfied.

2. Properties $P2$ and $P1$ are satisfied because $m \geq k > 1$, we obtain N_{k-1} as the point on $Seg(N_k, c_{k-1})$, that is 2^{k-1} away from c_{k-1}.
3. max is defined for $trail_P$, when $trail_P$ is created or updated starting from C. When max is (re)defined for $trail_P$, c_{max} is the position of the object and max is set to $(\lceil log_2(d_p) \rceil) - 1$. Thus the update algorithm satisfies property $P4$. □



(a) Move 1 (b) Move 2

Fig. 2. Updating $trail_P$

Definition 4 (Trail Stretch Factor). *Given $trail_P$ to an object p, we define the trail stretch factor for any point x on $trail_P$ as the ratio of the length along $trail_P$ from x to p, to the Euclidean distance $dist(x, p)$.*

Lemma 2. *The maximum* Trail Stretch Factor *for any point along $trail_P$, denoted as TS_p is $sec(\alpha) * sec(\frac{\alpha}{2})$ where $\alpha = arcsin(\frac{1}{2^b})$.*

Proof Sketch: We first show that the maximum *Trail Stretch Factor* occurs when point $N_{max}..N_1$ lie on a logarithmic spiral with origin p and the angle between the radius of the spiral and tangent to any point on the spiral being equal to $\alpha = arcsin(\frac{1}{2^b})$ [11]. We then use the property of logarithmic spirals that the ratio of length along spiral from any point on the spiral to the origin over the Euclidean distance of that point to the origin is $sec(\alpha)$. □

Lemma 3. *The length of $trail_P$ for an object P starting from a level $k(0 < k \leq max)$ vertex, denoted as L_k is bounded by $(2^k + 2^{k-b}) * TS_p$.*

Proof Sketch: $dist(c_k, p) < 2^{k-b}$. Therefore, $dist(N_k, p) < 2^k + 2^{k-b}$. Then using lemma 2, the result follows. □

Theorem 1. *The upper bound on the amortized cost of updating $trail_P$ when object P moves distance $d_m(d_m > 1)$ is $4 * (2^b + 1) * TS_p$.*

Proof. Note that in *update* whenever $trail_P$ is updated starting at the level k vertex, we set $c_{k-1} = p$. P can now move a distance of 2^{k-1-b} before another update starting at the level k vertex. Thus, between any two successive updates starting from a level k vertex, the object must have moved at least a distance of 2^{k-1-b}. The total to cost to create a new path and delete the old path starting from a level k vertex costs at most $2 * L_k$.

Note that over a distance d_m where $d_m > 1$, the *update* can start at level $((\lfloor log_2(d_m) \rfloor) + (b+1))$ vertex at most once. This is because, when update starts at level $((\lfloor log_2(d_m) \rfloor) + (b + 1))$ vertex, only when P has moved at least $2^{((\lfloor log_2(d_m) \rfloor)+(b+1)-1-b)} = d_m$ distance. Similarly, update can start at level $(b + 1)$ vertex atmost d_m times, update can start at level $(b + 2)$ vertex can at most $\lfloor d_m/2 \rfloor$ times, and so on. Adding the total cost, Theorem 1 follows. □

b	Trail Stretch	*Update* Cost
1	1.2	$14 * d_m * logd_m$
2	1.05	$20 * d_m * logd_m$
> 3	Approaches 1	$4 * (2^b + 1) * d_m * logd_m$

Fig. 3. Effect of b on *Update* Cost

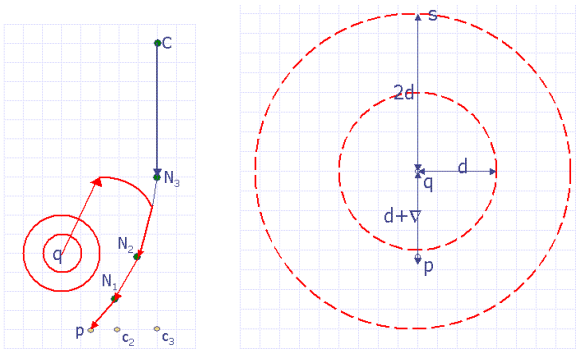
For illustration purposes, we summarize the *Trail Stretch factor* and *update* costs for different values of b in Fig. 3. We explain the significance of the refinement of *Trail* by varying b in Section 5.

3.3 Basic Find algorithm

Given $trail_P$ exists for an object P in the network, we now describe a basic $find$ algorithm that is initiated by object Q at point q on the plane. We use a basic ring search algorithm to intersect $trail_P$ starting from Q in a distance sensitive manner. We then show from the properties of the *Trail* tracking structure that starting from this intersection point, the current location of P is reached in a distance sensitive manner.

Basic $find$ Algorithm:

1. With center q, successively draw circles of radius $2^0, 2^1, ...2^{\lfloor log(d_{qC}) \rfloor - 1}$, until $trail_P$ is intersected.
2. If $trail_P$ is intersected, follow it to reach object P; else follow $trail_P$ from C (note that if object exists, $trail_P$ will start from C).



(a) $find$ Path (b) Farthest *Find* Point

Fig. 4. Basic Find Algorithm in *Trail*

Theorem 2. *The cost of finding an object P at point p from object Q at point q is $O(d_f)$ where d_f is $dist(p, q)$.*

Proof. Note that as q is distance d_f away from p, a circle of radius $2^{\lceil log(d_f) \rceil}$ will intersect $trail_P$. Hence the total length traveled along the circles before intersecting $trail_P$ at point s is bounded by $2 * \pi * \sum j = 1^{\lceil log(d_f) \rceil} 2^j$, i.e., $8 * \pi * d_f$. The total cost of connecting segments between the circles is bounded by $2 * d$.

Now, when the trail is intersected by the circle of radius $2^{\lceil log(d_f) \rceil}$, the point s at which the trail is intersected can be at most $3 * d_f$ away from the object q. This is illustrated in Fig. 4(b). In this figure, q is $d_f + \delta$ away from p. Hence the trail can be missed by circle of radius 2^{d_f}. From lemma 3, we have that distance along the trail from s to p is at most $3 * TS_p * d_f$. Thus, the cost of finding an object P at point p from object Q at point q is $O(d_f)$ where d_f is $dist(p, q)$. □

4 Implementing Trail in a WSN Grid

In this section, we refine the *Trail* protocol described for a continuous 2-d plane so as to implement it in a 2-d discrete WSN grid and describe its robustness properties. Each node is assigned some grid location x, y and is aware of that location. We refer to unit distance as the one hop communication distance. $dist(i, j)$ now stands for distance between motes i and j in these units. We also assume the existence of an underlying geographic routing protocol such as GPSR [12], aided by an underlying neighborhood service that maintains a list of neighbors at each mote. In the WSN grid, we assume that nodes in the network can fail due to energy depletion or hardware faults and there can be a bounded error in the placement of motes with respect to their ideal grid locations, thus leading to *holes* in the network. However, we assume that the network may not be partitioned; there exists a path between every pair of nodes in the network.

When implementing on a WSN grid, *Trail* is affected by the following factors:(1) discretization of points to nearest grid location; (2) Overhead of routing between any 2 points on the grid; and (3) *holes* in the network. We discuss these issues in this section.

Analysis of Routing Stretch Factor: When using geographic routing to route on a grid, the number of hops to communicate across a distance of d units will be more than d. We measure this stretch in terms of the routing stretch factor, defined as the ratio of the communication cost (number of transmissions) between any two grid locations, to the euclidean distance d between two grid locations. It can be shown that the upper bound on the routing stretch factor for the WSN unit grid is $\sqrt{2}$. The routing stretch factor will decrease in the denser grids because there are more nodes and routes will be increasingly closer to the segment between two grid points. We analyze the impact of failures on the routing stretch factor in Section 6.

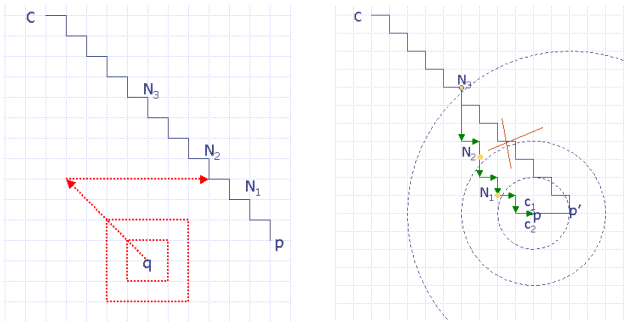
4.1 Implementing *find* on WSN Grid

We now describe how to implement the *find* algorithm in the WSN grid. As seen in Section 3, during a find, exploration is performed using circles of increasing radii around the finder. However, in the grid model, we approximate this procedure and instead of exploring around a circle of radius r, we explore along a square of side $2 * r$. The perimeter of the square spans a distance $8 * r$ instead of $2 * \pi * r$. We could use tighter approximations of the circle, but approximating with a square is simple for a grid.

Lemma 4. *The upper bound on the cost of finding an object P at point p from object Q at point q is $38 * d$ where d is $dist(p, q)$.*

4.2 Implementing *Update* on WSN Grid

We use three types of messages in the update actions. Initially, when an object is detected at a node, it sends an *explore* message that travels in around the square perimeters of increasing levels until it meets $trail_P$ or it reaches the center. Note that if the object is updated continuously as it moves, then the *explore* message will intersect the trail within a 1 hop distance. As before, the trail update is started from the level m vertex node where m is the minimal index such that $dist(c_m, p) < 2^{m-1}$ for all j such that $max \geq j \geq m$.



(a) *find* in a WSN grid (b) *update* in a WSN grid

Fig. 5. Find and Update Algorithm in a WSN grid

Starting from the level m node where update is started, a new path is created by sending a *grow* message towards c_{m-1}. Geographic routing is used to route the message towards c_{m-1}. On this route, the node closest to, but outside a circle of radius 2^{m-1} around c_{m-1} is marked as N_{m-1}. This procedure is then repeated at subsequent vertex motes and the path is updated. Fig. 5(b) shows how a trail is updated in the grid model with the grid spacing set equal to the unit communication distance. The vertex pointers $N_3, ...N_1$ are shown approximated on the boundary of the respective circles. Also, starting from the level k node where update is started, a *clear* message is used to delete the old path. We formally state the *update* and *find* algorithms in guarded command notation that for reasons of space have been relegated to our anonymous technical report [11]. We also implement the algorithms in *Java*, which we use in Section 6, to study the performance of *Trail*.

4.3 Fault-Tolerance

Due to energy depletion and faults, some nodes may fail leading to *holes* in the WSN grid. A hole consists of a contiguous set of nodes that have failed in the network. *Trail* uses minimal infrastructure and does not require expensive constructions such as hierarchical partitioning and in contrast to such solutions that are vulnerable to failures of nodes higher in the hierarchy, *Trail* supports a graceful degradation in performance in the presence of mote failures. We discuss the robustness of *Trail* under three scenarios: during *update*, maintaining an existing trail and during *find*.

Tolerating node failures during *update*: A *grow* message is used to update a trail starting at a level k mote and is directed towards the center of circle $k-1$. In the presence of holes, we use a right hand rule, such as in [12], in order to route around the hole and reach the destination. As indicated in the *update* algorithm for WSN grid, during routing the node closest to, but outside a circle of radius 2^{k-1} around c_{k-1} is marked as N_{k-1}. Since we assume that the network cannot be partitioned, eventually such a node will be found. (If all nodes along the circle have failed, the network is essentially partitioned).

Maintaining an existing trail: Nodes may fail after a trail has been created. Also, in some cases, *clear* messages may fail thereby not deleting an old trail. In order to stabilise from these faulty states, we use periodic *heartbeat* actions along the trail. We state the stabilizing actions in guarded command notation and explain how they restore the invariants. For reasons of space, we have relegated this discussion to the technical report.

Tolerating failures during a *find*: We now describe how the *find* message explores in squares of increasing levels. When an *explore* message comes across a hole, it is rerouted only radially outwards of the square and we do not allow back tracking. If all nodes in the forward direction of the explore have failed, then the *level* of search is incremented and routed towards a node in the next level. Thus, in the presence of larger holes, we abandon the current level and move to the next level, instead of routing around the hole back to the current level of exploration. Finally, if even that fails, the destination is marked as C and message is routed towards C. In the worst case, *find* may reach C.

5 Refinements to Trail

In this section, we discuss two techniques to refine the basic *Trail* network protocol: (1) tuning how often to update a *Trail* tracking structure, and (2) tuning the shape of a *Trail* tracking structure.

5.1 Tightness of *Trail* Tracking Structure

The frequency at which $trail_P$ is updated depends on parameter constant b in property $P3$ of $trail_P$. As seen in Section 3, for values of $b > 1$, $trail_P$ is updated more and more frequently, hence leading to larger update costs. However, $trail_P$ becomes more tighter and tends to a straight line with the *trail stretch factor* approaching 1. We exploit this tightness of $trail_p$ to optimize the *find* strategy.

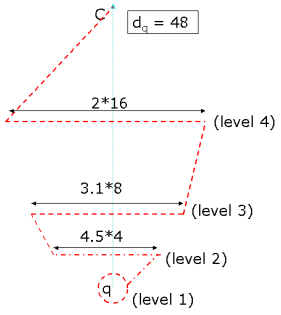


Fig. 6. Optimized *find*

The intuition behind this optimization is that since the trail to any object P originates at C, the angle formed by p with C and the higher level vertices is small and bounded. Hence as the levels of explorations increase in *find*, we can progressively decrease the size of exploration from full circles to cones of smaller angles. As an example, when $b = 2$, we prove that at the three highest levels of search, a conical pattern of search as shown in Fig. 6 is sufficient to guarantee distance sensitivity. In Fig. 6, $b = 2$, the object q is at distance 48 units from C. The levels of exploration are in the range 0..4. Exploration is along circles until level 1 and then along cones at levels 2 to 4. By increasing b, the angles formed by P with C and vertices $N_{max}..N_1$, start getting smaller. Therefore as b increases, the number of levels at which this optimization can be performed increases [11].

Impact of the Optimization: When using the optimized *find* strategy, the upper bound on $find(P, Q)$ costs remains the same when d_{pq} is small. However, we exploit the facts that trails to objects converge at C and therefore decrease the size of exploration at higher levels of search. Hence the upper bound costs for *find* decrease when d_{pq} increases. In other words, when d_{pq} is large, we mitigate the cost of Q having to explore at the lower levels. As an example, when $b = 2$ we show that the upper bound *find* costs decrease from $38 * d$ to $14 * d$ as d_{pq} increases [11].

The optimization of *find* at higher levels is thus significant in that it yields: (1) smaller upper bounds for objects that are far away from the finder; and (2) lower average cost of $find(p, q)$ over all possible locations of q and p. We note that there

are limits to tuning the frequency of updates, because for extreme values of b distance sensitivity may be violated. For example, for large values of b, that cause $dist(p, c_k) < y$ where y is a constant we end up with having to update the entire $trail_P$ when an object moves only a constant distance y. Similarly, for values of $b < 0$, the *Trail Stretch Factor* becomes unbounded with respect to distance from an object. Thus an object could be only δ away from a point on $trail_P$, yet the distance along $trail_P$ from this point to the p could travel across the network.

5.2 Modifying Trail Segments

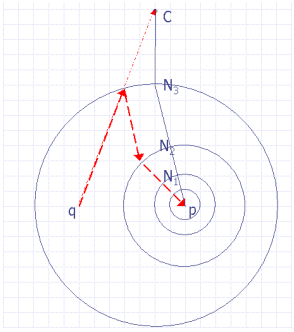


Fig. 7. Find Centric Trail

A second refinement to *Trail* is by varying the shape of the tracking structure by generalizing property $P2$ of $trail_P$. Instead of *trail segment* k between vertex N_k and N_{k-1} being a straight line, we relax the requirement on *trail segment* k to be of length at most $(2 * \pi + 1) * 2^k$. By publishing information of P along more points, the *find* path can be more straight towards C. An extreme case is when *trail segment* k is a full circle of radius 2^k centered at c_k and $segment(N_k, N_{k-1})$. We call this variation of *Trail* the *Find-centric Trail*.

Find-Centric Trail: In this refinement, the *find* procedure eschews exploring the circles (thus traversing only straight line segments) at the expense of the update procedure doing more work. This alternative data structure is used when objects are static or when object updates are less frequent than that of *find* queries in a system. Let $trail_P$ for object P consist of segments connecting $C, N_{max}, .., N_1, p$ as described before and, additionally, let all points on the circles $Circ_k$ of center c_k and radius 2^k contain pointers to their respective centers, where $max \geq k > 0$.

Starting at q, the *find* path now is a straight line towards the center. If a circle with information about object P is intersected then, starting from this point, a line is drawn towards the center of the circle. Upon intersecting the immediate inner circle (if there is one), information about its respective center is found, with which a line is drawn to this center. Object P is reached by following this procedure recursively.

Lemma 5. *In* Find Centric Trail, *when* $b = 1$, *the total cost of finding an object* P *at point* p *from object* Q *at point* q *is* $14 * d_f$ *where* $d_f = dist(p, q)$. □

When objects are static, we define the cost of querying for object P as simply the cost of reaching information about P. Since objects are static, we need not follow the pointers all the way to P.

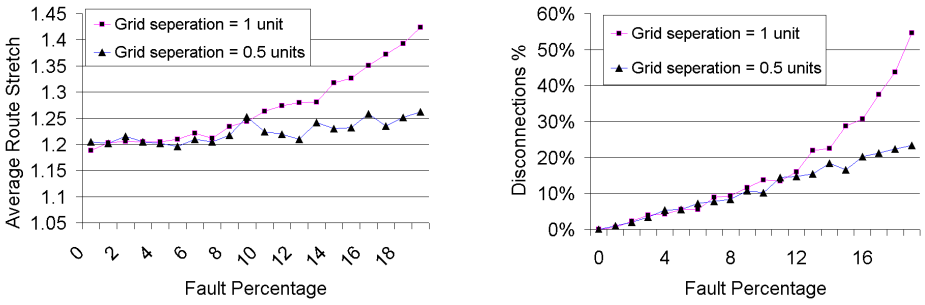
Lemma 6. *In* Find Centric Trail, *when objects are static, the upper bound on the cost of querying for information about an object* P *at point* p *from object* Q *at point* q *is* $2 * d_f$ *where* $d_f = dist(p, q)$. □

6 Performance Evaluation

In this Section, we evaluate the performance of *Trail* using simulations in *JProwler* [13]. The goals of our simulation are: (1) to study the effect of routing stretch and discretization errors on the trail stretch factor, (2) to study the effect of uniform

node failures on the performance of *Trail* and (3) to compare the average costs for $find$ and *update*, as opposed to the upper bounds we derived earlier. Our simulation involves a 90 by 90 Mica2 mote network arranged on a grid. We implement geographic routing on a grid to route messages in the network. In the presence of failures we use a left hand rule to route around the failure [12]. We assume an underlying link maintenance layer because of which the list of *up* neighbors is known at each node.

Routing Stretch: We first study the effect of *holes* in the network on the routing stretch factor. We simulate two different density models and inject node failures from 1% to 20% that are uniformly distributed across the network. We consider a grid separation of unit distance and 0.5 unit distance. We randomly select any two points in the network and measure the average routing stretch factor to route between 300 such pairs. From Fig. 8(a), we see that the routing stretch factor is a small constant factor over the actual distance between two nodes. The stretch factor decreases as expected when density increases. As the fault percentages increase, the number of disconnections in the network increase. The average route stretch factors shown are for the instances when the network is actually connected, and in these instances the average stretch factor does not increase significantly.



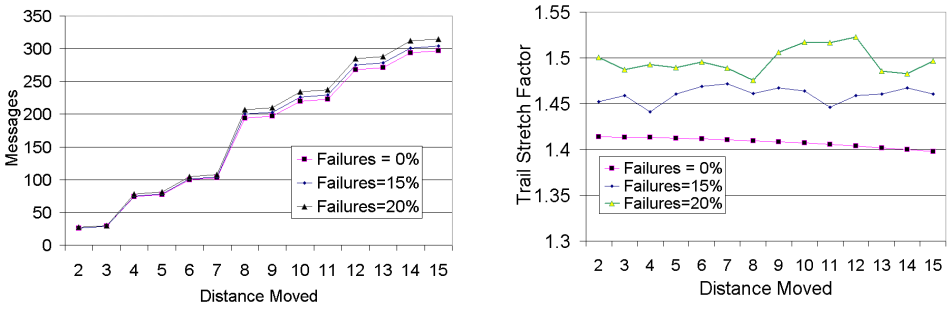
(a) Routing Stretch Factor

(b) Disconnection %

Fig. 8. Routing Stretch Factor in a Grid Network

Performance of Update Operations: We determine the number of messages exchanged for object updates over different distances when an object moves continuously in the network. We consider the unit grid separation, where each node has at most 4 communication neighbors. The number of neighbors may be lesser due to failures. We calculate the amortized cost by moving an object in different directions and then observing the cumulative number of messages exchanged up to each distance from the original position to update the tracking structure. The results are shown in Fig. 9(a). The jumps visible at distances 4 and 8 show the *logd* factor in the amortized cost. At these distances, the updates have to be propagated to a higher level. We also study the effect of uniform failures in the network on the increase in update costs. We consider fault percentages upto 20. We see from the figure that even with failures the average communication cost increases log linearly. This indicates that the failures are handled locally.

Trail Stretch Factor: From Section 3, we note that in the continuous model, for an object P at distance d_{pC} from C, $trail_P$ is less than $1.2 \times d_{pC}$. We now study the effect of routing overhead and the discretization factor on the length of the tracking



(a) Trail Update Cost (Amortized) (b) Trail Stretch Factors

Fig. 9. Trail Update Costs and Trail Stretch

structure that is created. We measure the trail length in terms of the number of hops along the structure. Fig. 9(b) shows the average ratio of distance from C to the length of the trail during updates over different distances from the original position. When the trail is first created, the trail stretch is equal to the routing stretch from C to the original location. In the absence of failures, we notice that the trail stretch increases to 1.4 at updates of smaller distances and then starts decreasing. This can be explained by the fact the trail for an object starts bending more uniformly when the update is over a large distance. Even in the presence of failures, the trail stretch factor increases to only about 1.6 times the actual distance.

Performance of $find$:

We now compare the average $find$ costs with upper bounds derived. We fix the finder at distance 40 units from C. We vary the distance of object being found from 2 to 16. We evaluate using the basic $find$ algorithm with $b = 1$ and the optimized $find$ algorithm discussed in Section 5 using $b = 2$. In the optimized $find$, at levels 2 to 4, we do not explore the entire circle. The results are shown in Fig. 10; the upper bound $38 * d$ is indicated using dotted lines and we find that the number of messages exchanged during *find* operations are significantly lower. The jumps at distances 3, 5 and 9 are due to increase in levels of exploration at these distances.

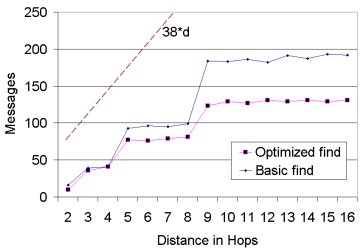


Fig. 10. Trail: Average find Cost

Experimental Evaluation: We have implemented *Trail* for the special case of a long linear topology network for demonstarting an intruder interceptor tracking application. We have experimentally validated the performance of this version of *Trail* on 105 Mica2 motes under different scaling factors such as the number of objects in the system, the frequency of queries, and the speed of the objects in the network. For reasons of space, these results are relegated to a technical report [11].

7 Related Work

In this section, we discuss related work and also compare the performance of *Trail* with other protocols designed for distance sensitive tracking and querying.

Tracking: As mentioned earlier, mobile object tracking has received significant attention [3, 4, 6] and we have focused our attention on WSN support for tracking. Some network tracking services [2] have nonlocal updates, where update cost to a tracking structure may depend on the network size rather than distance moved. There are also solutions such as [1, 3, 4] that provide distance sensitive updates and location.

Locality Aware Location Services *(LLS)* [1] is a distance sensitive location service designed for mobile adhoc networks. In LLS, the network is partitioned into hierarchies and object information is published in a spiral structure at well known locations around the object, thus resulting in larger update costs whenever an object moves. The upper bound on the update cost in LLS is $128 * d_m * logd_m$, where d_m is the distance an object moves, as opposed to the $14 * d_m * logd_m$ cost in *Trail*; the upper bounds on the $find$ cost are almost equal. Moreover, as seen in Section 5, we can further reduce the upper bound on the $find$ cost at higher levels in *Trail*.

The Stalk protocol [4] uses hierarchical partitioning of the network to track objects in a distance sensitive manner. The hierarchical partitioning can be created with different dilation factors ($r \geq 3$). For $r = 3$ and 8 neighbors at each level, at almost equal $find$ costs, *Stalk* has an upper bound update cost of $96 * d * logd$ and this increase occurs because of having to query neighbors at increasing levels of the partition in order to establish *lateral* links for distance sensitivity [4].

	Update Cost	Find Cost	Size
Trail	14*d*logd	38*d	1.2*d_c
LLS	128*d*logd	36*d	16*D
Stalk	96*d*logd	39*d	3*D
Awerbuch Peleg	O(d*logd*logN)	16*d*log(2N)	4*D

D – n/w Diameter ; N – No. of Nodes ; d_c – Distance from C

Fig. 11. Trail: Analytical Comparison

Both *Stalk* and *LLS* use a partitioning of the network into hierarchical clusters which can be complex to implement in a WSN, whereas *Trail* is cluster-free. Moreover, in *Stalk*, the length of the tracking structure can span the entire network as the object keeps moving and, in *LLS*, the information about each object is published in a spiral structure across the network. In comparison, *Trail* maintains a tighter tracking structure (i.e., with more direct paths to the center) and is thus more efficient and locally fault-tolerant.

In [3], a hierarchy of regional directories is constructed and the communication cost of a find for an object d_f away is $O(d_f * log2N)$ and that of a move of distance d_m is $O(d_m * logD * logN)$ (where N is the number of nodes and D is the network diameter). A topology change, such as a node failure, however, necessitates a global reset of the system since the regional directories depend on a non-local clustering program that constructs sparse covers.

Querying and storage: Querying for events of interest in WSNs has also received significant attention [14–16] and some of them focus on distance sensitive querying. We note that *Trail*, specifically the *Find-centric* approach can also be used in the context of static events.

Distance Sensitive Information Brokerage [17] protocol performs a hierarchical clustering of the network and information about an event is published to neighboring clusters at each level. DSIB has a querying cost of $4 * d$ to reach information about an event. Using *Find-centric Trail* we can query information about a static event

at a cost of $2 * d$. We also note that when events are static, the publish strategy can be further optimized and we study this in a recent work.

Geographic Hash tables [15] is a lightweight solution for the in-network-querying problem of static events. The basic *GHT* is not distance sensitive since it can hash the event information to a broker that is far away from a subscriber. The distance sensitivity problem of GHT can be alleviated to an extent by using geographically bounded hash functions at increasing levels of a hierarchical partitioning as used in DIFS protocol. Still, attempting such a solution suffers from a multi-level partitioning problem: a query event pair nearby in the network might be arbitrarily far away in the hierarchy. However, we do note that *GHT* provides load balancing across the network, especially when the types of events are known and this is not the goal of *Trail*.

In [16], a balanced push-pull strategy is proposed that depends on the query frequency and event frequency; given a required query cost, the advertise operation is tuned to do as much work as required to satisfy the querying cost. In contrast, *Trail* assumes that query rates depend on each subscriber (and potentially on the relative locations of the publisher and subscriber), and it also provides distance sensitivity during find and move operations, which is not a goal of [16]. In directed diffusion [14], a tree of paths is created from all objects of interest to the tracker. All these paths are updated when any of the objects move. Also, a controller initiated change in assignment would require changing the paths. By way of contrast, in *Trail*, we impose a fixed tracking structure, and tracks to all objects are rooted at one point. Thus, updates to the structure are local and any object can find the state of any other object by following the same tracking structure. Rumor routing [18] is a probabilistic algorithm to provide query times proportional to distance; the goal of this work is not to prove a deterministic upper bound. Moreover, its algorithm does not describe how to update existing tracks locally and yet retain distance sensitive query time when objects move.

8 Conclusions and Future Work

We have presented *Trail*, a family of protocols for distance sensitive distributed object tracking in WSNs. *Trail* avoids the need for hierarchical partitioning by determining anchors for the tracking paths on-the-fly, and is more efficient than other hierarchy based solutions for tracking objects: it allows 7 times lower updates costs at almost equal *find* costs and can tolerate faults more locally as well.

Importantly, *Trail* maintains tracks from object locations to only one well-known point, the center of the network, which we claim is necessary to minimize the total track length for objects. Well-known points are necessary for distance sensitive tracking and, as we prove in the associated technical report of this paper [11], multiple well-known points cannot yield shorter total track length of objects. Moreover, since its tracks are *almost* straight to the center with a stretch factor close to 1, *Trail* tends to achieve the lower bound on the total track length. By using a tight tracking structure, *Trail* also able to decrease the upper bound $find$ costs at larger distances and thereby decrease the average find cost across the network.

We have shown that refinements of the basic *Trail* protocol are well suited for different network sizes and query frequency settings. We have validated the distance

sensitivity and fault tolerance properties of *Trail* in a simulation of 90 by 90 network using *JProwler*. We have also succesfully implemented a *Trail* protocol in the context of a pursuer evader application for a medium size (over 100 node) mote network.

Trail operates in an environment where objects can generate updates and queries asynchronously. We note that in such an environment, due to the occurrence of collisions, there can be an increase in the message complexity for querying and updates especially when the objects are densely located in the network. As future work, we are considering a *push* version of the network tracking service where snapshots of objects are published to subscribers in a distance sensitive manner, both in time and information, in order to increase the reliability and energy efficiency of the service when the density of objects in the network is high.

References

1. I. Abraham, D. Dolev, and D. Malkhi. LLS: A locality aware location service for mobile ad hoc networks. *DIALM-POMC*, 2004.
2. S. Dolev, D. Pradhan, and J. Welch. Modified tree structure for location management in mobile environments. In *INFOCOM*, pages 530–537, 1995.
3. B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the Associsation for Computing Machinery*, 42:1021–1058, 1995.
4. M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *OPODIS*, 2004.
5. A. Arora, P. Dutta, and S. Bapat et al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks, Special Issue on Military Communications Systems and Technologies*, 46(5):605–634, July 2004.
6. T. He, S. Krishnamurthy, and J. Stankovic et al. Vigilnet:an integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks*, 2004.
7. A. Arora and R. Ramnath et al. Exscal: Elements of an extreme wireless sensor network". In *The 11th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2004.
8. H. Cao, E. Ertin, and V. Kulathumani et al. Differential games in large scale sensor actuator networks. In *Information Processing in Sensor Networks (IPSN)*, 2006.
9. B. Sinopoli and C. Sharp et al. Distributed control applications within sensor networks. In *Proceedings of the IEEE*, volume 91, pages 1235–46, Aug 2003.
10. J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target indentities in wireless ad hoc networks. In *IPSN*, 2003.
11. Trail:. Technical Report http://briefcase.yahoo.com/trail_ewsn (pwd: 123456), 2005.
12. B. Karp and H. T. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of MobiCom*, 2000.
13. Vanderbilt University. JProwler. `http://www.isis.vanderbilt.edu/Projects/nest/jprowler/index.html`.
14. C. Intanogonwiwat and R. Govindan et al. Directed diffusion for wireless sensor networking. *IEEE Transactions on Networking*, 11(1):2–16, 2003.
15. S. Ratnasamy and B. Karp et al. GHT: A geographic hash table for data-centric storage. In *Wireless Sensor Networks and Applications (WSNA)*, 2002.
16. X. Liu, Q. Huang, and Y. Zhang. Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks. In *ACM Sensys*, 2004.
17. S. Funke and L. Guibas et al. Distance sensitive information brokerage in sensor networks. In *DCOSS*, 2006.
18. D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *ICDCS*, 2002.