Trail: A Distance-Sensitive Sensor Network Service For Distributed Object Tracking

VINODKRISHNAN KULATHUMANI, ANISH ARORA,
and MUKUNDAN SRIDHARAN
The Ohio State University
and
MURAT DEMIRBAS
University at Buffalo, SUNY

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 article, we present a wireless sensor network protocol, Trail, that supports distance-sensitive tracking of mobile objects for in-network subscribers upon demand. Trail achieves a find time that is linear in the distance from a subscriber to an object, via a distributed data structure that is updated only locally when the object moves. Notably, Trail does not partition the network into a hierarchy of clusters and clusterheads, and as a result Trail has lower maintenance costs, is more locally fault tolerant, and it better utilizes the network in terms of load balancing and minimizing the size of the data structure needed for tracking. Moreover, Trail is reliable and energy efficient, despite the network dynamics that are typical of wireless sensor networks. Trail can be refined by tuning certain parameters, thereby yielding a family of protocols that are suited for different application settings such as rate of queries, rate of updates, and network size. We evaluate the performance of Trail by analysis, simulations in a 90×90 sensor network, and experiments on 105 Mica2 nodes in the context of a pursuer-evader control application.

Categories and Subject Descriptors: C.2.1 [**Computer Communication Networks**]: Nework Architecture and Design—*Distributed networks, wireless communication*; C.2.2 [**Computer Communication Networks**]: Network Protocols

General Terms: Algorithms, Design, Reliability, Performance

Additional Key Words and Phrases: Distributed tracking, network protocol, applications of sensor actuator networks, data storage and query, fault tolerance, scalability, energy efficiency

This article is an extended version of the paper that appeared in the 2007 *Proceedings of the European Conference on Wireless Sensor Networks (EWSN'07)*.

Authors' addresses: V. Kulathumani (contact author), A. Arora, and M. Sridharan, Department of Computer Science and Engineering, The Ohio State University; email: vinodkri@cse.ohio-state.edu; M. Demirbas, Department of Computer Science and Engineering, University at Buffalo, SUNY.

DOI 10.1145/1498915.1498921 http://doi.acm.org/10.1145/1498915.1498921

ACM Reference Format:
Kulathumani, V., Arora, A., Sridharan, M., and Demirbas, M. 2009. Trail: A distance-sensitive sensor network service for distributed object tracking. ACM Trans. Sensor Netw. 5, 2, Article 15 (March 2009), 40 pages. DOI = 10.1145/1498915.1498921 http://doi.acm.org/10.1145/1498915.1498921

1. INTRODUCTION

Tracking of mobile objects has received significant attention in the context of cellular telephony, mobile computing, and military applications [Abraham et al. 2004; Dolev et al. 1995; Awerbuch and Peleg 1995; Demirbas et al. 2004]. In this article, we focus on the tracking of mobile objects using a network of static wireless sensors. Examples of such applications include those that monitor objects [Arora et al. 2004; He et al. 2006; Arora and Ramnath 2004], 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 wireless sensor network (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 source of the query may be close to the object itself but still have to communicate all the way to a central location. Such a centralized 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 [Cao et al. 2006] 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; that is, 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 object and 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; that is, the cost of an update depends only the distance moved by the object. Querying, by itself, for events or information of interest in WSNs has received significant attention [Intanogonwiwat et al. 2003; Ratnasamy et al. 2002; Liu et al. 2004] and some of that research focuses on distance-sensitive querying [Ratnasamy et al. 2002; Sarkar et al. 2006; Funke et al. 2006]. In those solutions, information from a source is published across the network in such a way that queries can be resolved in a distance-sensitive manner, but the solutions do not address the update of the published information when an object moves. The focus of our work is on the tracking of mobile objects and we additionally require that update cost is distance sensitive.

Contributions. In this article, 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. Trail has a find cost that is linear ($O(d_f)$) in terms of the distance (d_f) of the subscriber from the object. To this end, Trail maintains a tracking data structure by propagating mobile object information only locally, and satisfying the distance-sensitivity requirement for the track updates. The amortized cost of updating a track when an object moves a distance d_m is $O(d_m * log(d_m))$.

A basic Trail protocol can be refined by tuning certain parameters, thus resulting in a family of Trail protocols. Appropriate refinements of the 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; that is, it 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. Moreover, Trail scales well to networks of all sizes. As network size decreases, Trail gradually eschews local explorations and updates, and thus increasingly centralizes update and find.

We evaluate the performance of Trail by simulations in a 90×90 sensor network and experiments on 105 Mica2 nodes in the Kansei testbed [Arora et al. 2006]. This implementation has been used to support a distributed intruder-interceptor tracking application where the goal of the interceptor is to catch the intruders as far away from an asset as possible.

Overview of Solution. Trail maintains tracks from each object to only one terminating point, namely, the center of the network C; these tracks are *almost* straight to the center, with a stretch factor of at most 1.2 times the distance to C. Note that if the track to an object P is required to be always a straight line from C to the current location of P, resulting in a stretch factor equal to 1, then every time the object moves, the track has to be updated starting from C. This would not be a distance-sensitive approach. Therefore, we form the track as a set of *trail segments* and update only a portion of the structure depending upon the distance moved. Thus, given a terminating point, Trail maintains a track with lengths from the terminating point that is almost close to minimum. This is important because 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.

Given the track to an object, a find operation explores along circles of exponentially increasing radii until the track is intersected and then follows the track to the current location of the object. The tracks maintained in Trail only contain pointers to the current location of an object and not the state information of the object. Publishing the current state (i.e., location) of the object along all points in the track will violate distance sensitivity of updates because every move of the object will result in updating the entire track. Following the trail of an object from any location leads to the current location of the object, which contains the state of the object. Yet Trail is distance sensitive in terms of the find in the sense that the total cost of reaching the track for an object, and following the track to reach the object, is proportional to the distance of the finder from the object. Note that a find explores along circles until encountering a radii that is at most half the distance of the finder to C and then it searches at C where the track is certain to be found. Thus, C serves as a worst-case landmark for finding objects in the network.

In our solution, we make some design decisions, like choosing a single point to terminate tracks from all points in the network and avoiding hierarchy in maintaining the tracks. In Section 7, we analyze these aspects of our solution and compare them with other possible approaches.

Organization of the Article. In Section 2, we describe the system model and problem specification. In Section 3, we design the basic Trail protocol for a 2D real plane. Then, in Section 4, we present an implementation of the basic Trail protocol for a 2D sensor network. 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 analyze some design decisions made in our solution and compare them with other possible approaches. In Section 8, we discuss related work, and in Section 9 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 and association 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 problem of detecting objects in the network and uniquely associating them with previous detections is thus orthogonal to the tracking service that we discuss in this article. Detection and association services can be implemented in a centralized [Sinopoli et al. 2003] or distributed [Shin et al. 2003] fashion; the latter approach would suit integration with the tracking service that we discuss in this work. Detection and association could also be enabled by the objects being *cooperative*, for example, being tagged with RFID and announcing their identifier. We assume that the underlying detections and associations are always correct.

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

Definition 2.2 (*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'.

We note that our network service does not assume knowledge of the motion model of objects being tracked, in contrast to some query services [Lu et al. 2005] and, as such, the scope of every query in our case is the entire network as opposed to a certain locality. Nor does it assume a bound on the number of querying objects in the network or any synchrony between concurrent queries.

Network model. To simplify our presentation, we first describe Trail in a 2D real continuous plane. We then refine the Trail protocol to suitably implement in a random connected deployment of a wireless sensor network. In this model, we impose a virtual grid on the random deployment and snap each node to its nearest grid location (x, y). Each node is aware of this location. We refer to unit distance as the one-hop communication distance. *dist*(i, j) now stands for distance between nodes i and j in these units. We describe this model in more detail and the implementation of Trail in this discrete model in Section 4.

Fault model. In the wireless sensor network, we assume that nodes can fail due to energy depletion or hardware faults, or there could be insufficient density at certain regions, 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.

3. TRAIL

In this section, we use geometric ideas to design Trail for a bounded 2D 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 plane. Let P be an object being tracked, and p denote its location on the plane. 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 $dist(p, C)$ increases. The endpoints of each trail segment serve as marker points to update the tracking structure when an object moves. The point from where the update is started depends on the distance moved. Only when P moves a sufficiently large distance is $trail_P$ updated all the way from C. We now formally define $trail_P$.

Definition 3.1 ($trail_P$). The tracking data structure for object P, denoted by $trail_P$, for $dist(p, C) \geq 1$ is a path obtained by connecting any sequence of points $(C, N_{mx}, ..., N_k, ..., N_1, p)$ by line segments, where $mx \geq 1$, and there exist auxiliary points $c_1..c_{mx}$ that satisfy the properties $(P1)$ to $(P3)$ to follow. mx is defined as $\lceil(log_2(dist(C, p_o)))\rceil - 1$, where p_o is the location of p when $trail_P$ was initialized or updated starting from C.

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 plane.

—(P1): $dist(c_k, N_k) = 2^k$, $(mx \geq k \geq 1)$.

—(P2): N_{k-1}, $(mx \geq k \geq 1)$, lies on $Seg(N_k, c_{k-1})$; N_{mx} lies on $Seg(C, c_{mx})$.

—(P3): $dist(p, c_k) < 2^{k-b}$, $(mx \geq k \geq 1)$ and $b \geq 1$ is a constant.

If $(dist(p, C) = 0)$, $trail_P$ is C; and if $(0 \leq dist(p, C) < 1)$, $trail_P$ is $Seg(C, p)$.

Observations about $trail_P$. From the definition of $trail_P$, we note that the auxiliary points $c_1..c_{mx}$ are used to mark vertices $N_1..N_{mx}$ of $trail_P$. $P1$ and $P2$ describe the relation between the auxiliary points and the vertices of $trail_P$. Given $trail_P$, points $c_1..c_{mx}$ are uniquely determined using $P1$ and $P2$. Similarly, given p and $c_1, ..c_{mx}$, $trail_P$ is uniquely determined. These properties are stated in the following lemmas.

LEMMA 3.2. *Given $trail_P$, points $c_1..c_{mx}$ are uniquely determined.*

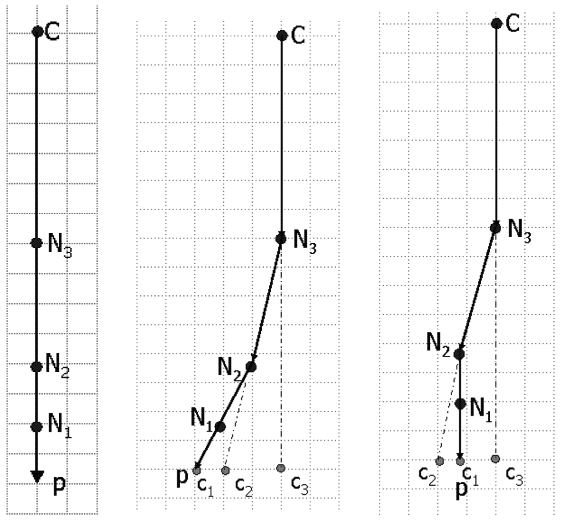
PROOF. Extend $Seg(C, N_{mx})$ of $trail_P$ by a distance of 2^{mx} to obtain c_{mx}. Similarly extend $Seg(N_k, N_{k-1})$ by 2^{k-1} to obtain c_{k-1} for $0 < k \leq mx$. Thus using properties $P1$ and $P2$ of $trail_p$, points $c_1..c_{mx}$ are uniquely determined, given $C, N_{mx}, .., N_1, p$ of $trail_P$. □

LEMMA 3.3. *Given $c_1, ...c_{mx}$ and p, $trail_P$ is uniquely determined.*

PROOF. N_{mx} lies on $Seg(C, c_{mx})$ such that $dist(c_{mx}, N_{mx}) = 2^{mx}$. Similarly N_{k-1} lies on $Seg(N_k, c_{k-1})$ such that $dist(c_{k-1}, N_{k-1}) = 2^{k-1}$ for $1 < k \leq mx$. Thus $C, N_{mx}, .., N_1, p$ of $trail_P$ are uniquely determined. □

By property $P3$, the maximum separation between p and any auxiliary point c_k decreases exponentially as k decreases from mx to 1. When an object moves a certain distance away from its current location, $trail_P$ has to be updated from the smallest index k such that property $P3$ holds at all levels. By changing parameter b in property $P3$, we can tune the rate at which the tracking structure is updated. We discuss these refinements in Section 5.

Note from the definition of $trail_P$ that mx is defined as $\lceil(log_2(dist(C, p_o)))\rceil - 1$ where p_o was the location of the object when $trail_P$ was either created or updated from C. The value of mx, which denotes the number of trail segments



(a) Initial (b) P moves 3 units from c_3 (c) P moves back towards c_3

Fig. 1. Examples of trail to an object P.

in $trail_P$, depends on the distance of P from C. When $trail_P$ is first created, $c_1, ..., c_{mx}$ are initialized to location p_o, the number of levels mx is initialized to $\lceil(log_2(dist(C, p_o)))\rceil - 1$, and $trail_P$ is a straight line. The value of mx is updated when $trail_P$ has moved a sufficient distance to warrant an update of $trail_P$ all the way from C. The update (and create) procedure for $trail_P$ is described in more detail in the following subsection.

We now show three examples of the tracking structure in Figure 1. In this figure, $b = 1$. Figure 1(a) shows $trail_P$ when $c_3, ..c_1$ are collocated. When P moves away from this location, $trail_P$ is updated and Figure 1(b) shows an example of $trail_P$ where c_2, c_1 are displaced from c_3. In Figure 1(v), $dist(c_3, c_2) =$ 2 units, $dist(c_2, c_1) = 1$ unit, and p and $c1$ are collocated. Moreover, N_3 lies on $Seg(C, c_3)$, N_2 lies on $Seg(N_3, c_2)$ and so on. In Figure 1(c) we show an example of a zigzag 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, we find the minimal index m, along $trail_P$ such that $dist(p, c_j) < 2^{j-b}$ for all j such that $mx \geq j \geq m$ 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 $N'_{m-1}, ..., N'_1$ 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 next.

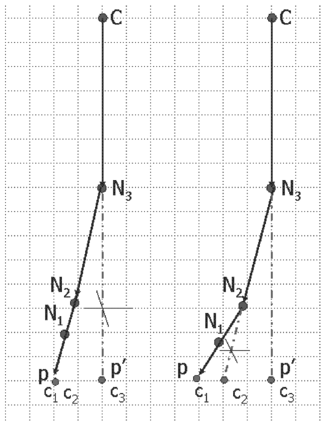


Fig. 2. Updating $trail_P$.

Algorithm. *Update*

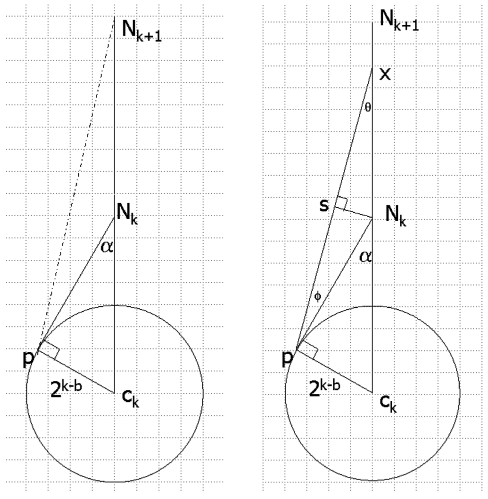
```
(1) Let $m$ be the minimal index on the trail such that $dist(p, c_j) < 2^{j-b}$ for all $j$ such that $mx \geq j \geq m$.
(2) $k = m$
(3) while $k > 1$
(a) $c_{k-1} = p$; Now obtain $N_{k-1}$ using property $P2$ as follows: the point on segment $N_k, c_{k-1}$, that is $2^{k-1}$ away from $c_{k-1}$.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

If no indices exist such that $dist(p, c_j) < 2^{j-b}$, 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, mx is set to $(\lceil log_2(dist(C, p))\rceil) - 1$. c_{mx} is set to p. N_{mx} is marked on $Seg(C, p)$ at distance 2^{mx} from c_{mx}. Step 2 is executed with $k = mx$.

Figure 2 illustrates an update operation, when $b = 1$. In Figure 2(a), $dist(p, p')$ is 2 units. Hence, the 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 $Seg(N_3, c_2)$ and N_1 lies on $Seg(N_2, c_1)$. In Figure 2(b), P moves further one unit. Hence the 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 $Seg(N_2, c_1)$.

Note. The Trail update algorithm described before is for a continuous 2D plane. In this algorithm, we have stated the minimum level from which $trail_P$ is updated and described how the vertices and auxiliary points are updated starting from this level. In Section 4, we describe Trail protocol on a wireless sensor network. A formal specification of the algorithm, in the form of guarded command actions, is also provided in Section 4. □

LEMMA 3.4. *The update algorithm for Trail yields a path that satisfies* $trail_P$.



(a) max angle in Trail (b) analyzing stretch

Fig. 3. Analyzing trail stretch factor.

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 $mx \geq j \geq m$. 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) mx is defined for $trail_P$, when $trail_P$ is created or updated starting from C. When mx is (re)defined for $trail_P$, c_{mx} is the position of the object and mx is set to $(\lceil log_2(dist(C, p))\rceil) - 1$. □

Definition 3.5 (*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 3.6. *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. We prove Lemma 3.6 by using the following steps.

—*(Maximum Angle ($\angle pN_kc_k$))*. Let the maximum angle formed by p and c_k at N_k in $trail_P$ for $(mx \geq k \geq 1)$ be denoted as α. Refer to Figure 3(a). Recall from properties of $trail_P$ that $dist(N_k, c_k) = 2^k$ and $dist(p, c_k) < 2^{k-b}$. Note that $\angle pN_kc_k$ is maximum when $Seg(N_k, p)$ is tangent to a circle of radius 2^{k-b} and center c_k. Therefore we have the following condition.

$$\alpha < arcsin(\frac{1}{2^b}) \tag{1}$$

—*(Maximum Stretch at a Given Level k)*. Let x be any point on $trail_P$ which lies on $Seg(N_{k+1}, N_k)$. Refer to Figure 3(b). We note the following equation

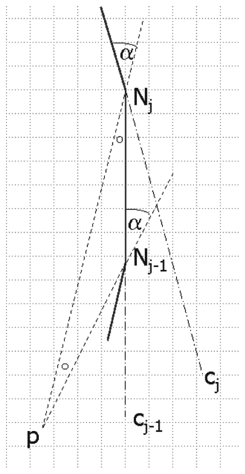


Fig. 4. Analyzing trail stretch.

based on the geometry of Figure 3(b).

$$\frac{(dist(x, N_k) + dist(N_k, p))}{dist(x, p)} = \frac{(sin(\theta) + sin(\phi))}{sin(\theta + \phi)} \tag{2}$$

Also note that $(\theta + \phi) = \angle pN_k c_k$. Using this, we get Eq. (3).

$$(\theta + \phi) \leq \alpha \tag{3}$$

Let $f(\theta, \phi)$ denote the following function.

$$f(\theta, \phi) = \frac{(sin(\theta) + sin(\phi))}{sin(\theta + \phi)} \tag{4}$$

It can be shown that $f(\theta, \phi)$, where $\theta > 0$, $\phi > 0$ and $\theta + \phi \leq \alpha$ is maximum when $\theta = \phi = \frac{\alpha}{2}$. We state this as Proposition A.1 and prove it in Appendix A. Substituting $(\theta = \phi)$ in Eq. (2), we get the following condition.

$$\frac{(dist(x, N_k) + dist(N_k, p))}{dist(x, p)} \leq sec(\frac{\alpha}{2}) \tag{5}$$

Thus, we see that at a single level k the maximum stretch for $trail_P$ occurs when $\angle pN_k c_k$ is α for a point x on $Seg(N_{k+1}, N_k)$ such that $\angle pxN_k = \angle xpN_k$. Since $\angle pxN_k = \angle xpN_k$ when the maximum occurs, we also get the following equation.

$$\frac{(dist(x, N_k))}{dist(x, p)} = \frac{(dist(N_k, p))}{dist(x, p)} = \frac{sec(\frac{\alpha}{2})}{2} \tag{6}$$

—*(Maximum Trail Stretch Factor from Vertex N_k to p).* In order to find the maximum stretch factor over multiple levels, we consider $trail_P$ to be *split* at vertices N_k to N_2 in such a way that the stretch is maximized at each level. Thus we let $\angle pN_j c_j = \alpha$ and $\angle pN_j N_{j-1} = \angle N_j pN_{j-1}$ for all $k \geq j > 1$. In Figure 4, we show one such trail segment, $Seg(N_j, N_{j-1})$ of $trail_P$.

Using Eq. (6), we get the following equations.

$$\frac{dist(N_{j-1}, p)}{dist(N_j, p)} = \frac{sec(\frac{\alpha}{2})}{2} \qquad \forall j : k \geq j > 1 \tag{7}$$

$$\frac{dist(N_j, N_{j-1})}{dist(N_j, p)} = \frac{sec(\frac{\alpha}{2})}{2} \qquad \forall j : k \geq j > 1 \tag{8}$$

When the aforesaid configuration is repeated at all levels of $trail_P$, we determine the ratio of the lengths of two successive trail segments, $Seg(N_{j-1}, N_{j-2})$ and $Seg(N_j, N_{j-1})$. Using Eq. (7) and Eq. (8) we get the following equation.

$$\frac{dist(N_{j-1}, N_{j-2})}{dist(N_j, N_{j-1})} = \frac{sec(\frac{\alpha}{2})}{2} \qquad \forall j : k \geq j > 2 \tag{9}$$

Let L_k denote the length along $trail_P$ from vertex N_k. Using Eq. (9), we get

$$L_k = dist(N_2, N_1) * \sum_{j=0}^{j=(k-2)} (2 * cos(\frac{\alpha}{2})^j) + dist(N_1, p). \tag{10}$$

Let R_k denote the trail stretch factor from N_k.

$$R_k = \frac{L_k}{dist(N_k, p)} \tag{11}$$

Upon simplification, using Eqs. (7), (8), and (10) we get

$$R_k = \frac{1}{2 * cos(\frac{\alpha}{2}) - 1} + \frac{1}{(2 * cos(\frac{\alpha}{2}))^{k-1}} * \left(1 - \frac{1}{2 * cos(\frac{\alpha}{2}) - 1}\right). \tag{12}$$

Since $\alpha < \frac{\pi}{6}$ for $b \geq 1$, $0 < 2 * cos(\frac{\alpha}{2}) - 1 \leq 1$. Therefore we get

$$R_k \leq \frac{1}{2 * cos(\frac{\alpha}{2}) - 1}. \tag{13}$$

Since $cos(\alpha) = 2 * cos^2(\frac{\alpha}{2}) - 1$, $cos(\frac{\alpha}{2}) \leq 1$ and $0 < 2 * cos(\frac{\alpha}{2}) - 1 \leq 1$, we get

$$R_k \leq \frac{1}{cos(\alpha)}. \tag{14}$$

—*(Maximum Trail Stretch Factor from Any Point in $trail_P$ to p).* Let x be any point on $trail_P$ which lies on a level $k+1$ segment, namely $Seg(N_{k+1}, N_k)$, but is not a *vertex* point. Let L_{xp} denote the length along $trail_P$ from x to p. Using Eq. (14) and Eq. (5), we have the following inequalities.

$$\begin{aligned} L_{xp} &\leq dist(x, N_k) + dist(N_k, p) * sec(\alpha) \\ &\leq (dist(x, N_k) + dist(N_k, p)) * sec(\alpha) \\ &\leq sec\left(\frac{\alpha}{2}\right) * sec(\alpha) * dist(x, p) \end{aligned}$$

Thus we have proved that the maximum *trail stretch factor* for any point along $trail_P$, denoted as TS_p, is $sec(\alpha) * sec(\frac{\alpha}{2})$ where $\alpha = arcsin(\frac{1}{2^b})$. □

LEMMA 3.7. *The length of $trail_P$ for an object P starting from a level $k(0 < k \leq mx)$ 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 3.6, the result follows. □

THEOREM 3.8. *The upper bound on the amortized cost of updating $trail_P$ when object P moves distance $d_m (d_m \geq 1)$ is $4 * (2^b + 1) * TS_p * d_m * log(d_m)$.*

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 cost to create a new path and delete the old path starting from a level k vertex costs at most $2 * L_k$.

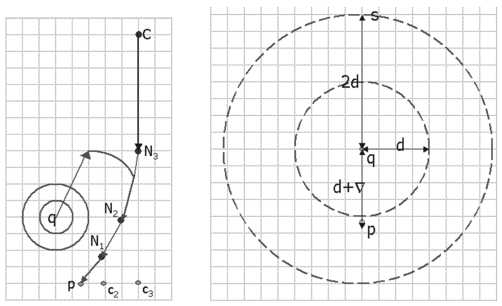
When an object moves a total distance d_m, where $d_m \geq 1$, it could involve multiple updates at smaller distances. The object could be detected at multiple instances over this distance d_m. Therefore we calculate the upper bound on the amortized cost of update when the object moves distance d_m. We consider the minimum distance to trigger an update to be 1 unit. Note that 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}. Thus over a distance d_m, update can start at a level $(b+1)$ vertex at most d_m times, update can start at a level $(b+2)$ vertex can at most $\lfloor d_m/2 \rfloor$ times, and so on. The update can start at a level $(\lfloor log_2(d_m) \rfloor + b + 1)$ vertex at most once. Adding the total cost, the result follows. □

Remark on update cost. We note the following points about the update cost characterized in Theorem 3.8. (1) First of all, over a distance of d_m, the object can be *detected* in multiple instances and consequently multiple updates of the track can result over a distance d_m, but the total cost of all the updates over a distance d_m is $O(d_m * log(d_m))$. We consider the minimum distance to trigger an update to be 1 unit. We add the total cost resulting from each update and the sum of the cost of *all* these possible updates results in an upper bound on the amortized cost, stated in Theorem 3.8. (2) Secondly, it is *only* the amortized cost of update which is of the order of $O(d_m * log(d_m))$ and distance sensitive. During a move of distance d_m by any object P, there could be updates triggered over a smaller distance d_{ms} (e.g., $d_{ms} = 1$), that result in $trail_P$ being updated from a much larger level $k >> 1$. But the update algorithm of Trail guarantees that whenever $trail_P$ is updated starting at the level k vertex, 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}. There can only be updates at levels smaller than k between two updates at level k. Upon adding up all these update costs, we get the amortized *update* cost, which is of the order of $O(d_m * log(d_m))$ and distance sensitive.

For illustration, we summarize the *trail stretch factor* and update costs for different values of b in Figure (5). We explain the significance of the refinement of Trail by varying b in Section 5.

b	Trail Stretch	*Update* Cost
1	1.2	$14 * d_m * logd_m$
2	1.05	$21 * d_m * logd_m$
> 3	Approaches 1	$4 * (2^b + 1) * d_m * logd_m$

Fig. 5. Effect of b on update cost.



(a) *find* path (b) Farthest *find* point

Fig. 6. Basic find algorithm in Trail.

3.3 Basic Find Algorithm

Given $trail_P$ exists for an object P, 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.

Algorithm. Basic Find

(1) With center q, successively draw circles of radius $2^0, 2^1, ...2^{\lfloor log(dist(q,C))\rfloor -1}$, 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).

THEOREM 3.9. *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 $2 * \pi * \sum_{j=1}^{\lceil log(d_f)\rceil} 2^j$, namely, $8 * \pi * d_f$. The total cost of connecting segments between the circles is bounded by $2 * d_f$.

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 p. This is illustrated in Figure 6(b). In this figure, q is $d_f + \nabla$ away from p. Hence the trail can be missed by circle of radius $2^{log(d_f)}$. From Lemma 3.6, 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 $dist(p, q)$. □

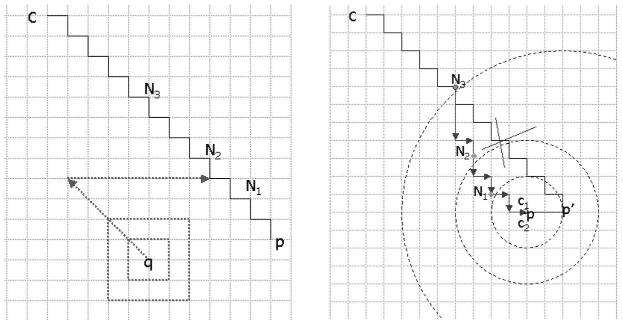
Update cost when consisting of discrete jumps. Note that the update costs characterized in Figure 5 are for the continuous update case when updates are performed after every unit distance of move. Now consider the case where a total move of distance d_m consists of multiple discrete jumps. In each jump, an object disappears at a certain location and is later detected at another location at distance d_x away such that $d_x >> 1$. In this case, we note that the cost of individual smaller updates need not be added. However, there is an additional cost of exploring to find an existing trail because we do not assume memory of the previous location of an object. (In the continuous setting where updates performed after every one unit of move, we ignore the cost of finding an existing $trail_P$, as it will exist within a distance of 1 unit.) But recall from the find algorithm that when P moves distance d_x away, an existing $trail_P$ will be intersected at an upper bound cost of $8 * \pi * d_x$, namely $O(d_x)$. Therefore the amortized update cost still remains $O(d_m * log(d_m))$ over a total distance d_m consisting of such discrete jumps.

4. IMPLEMENTING TRAIL IN A WSN

In this section, we describe how to implement the Trail protocol in a WSN that is a discrete plane, as opposed to a continuous plane as described in the previous section. Trail can be implemented in any random deployment of a WSN aided by some approximation for routing along a circle. We describe one such implementation in the following.

System model. Consider any random deployment of nodes in the WSN. We impose a virtual grid on this deployment and snap each node to its nearest grid location (x, y) and assume that each node is aware of this location. We refer to unit distance as the one-hop communication distance. $dist(i, j)$ now stands for distance between nodes i and j in these units. The separation along the grid is less than or equal to the unit distance. When the network is dense, the grid separation can be smaller. The neighbors of a node comprise a set of all nodes within unit distance of the node. Thus when the grid separation is unit distance, there are at most 4 neighbors for each node. We also assume the existence of an underlying geographic routing protocol such as GPSR [Karp and Kung 2000], aided by an underlying neighborhood service that maintains a list of neighbors at each node.

Note. We have implicitly assumed in the aforesaid model that each location on the grid is mapped to a unique node. This can be achieved by decreasing the size of the grid to be equal to the smallest separation between any 2 nodes in the network. However, this assumption is not necessary. In other words, all nodes need not be necessarily assigned to some location on the grid. Nodes can take turns playing the role of a given location. But for ease of exposition, we have abstracted away these possibilities in our model.



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

Fig. 7. Find and update algorithm in a WSN grid.

Fault model. We assume that nodes in the network can fail due to energy depletion or hardware faults, or there could be insufficient density at certain regions, 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. A node may also transiently fail. But we assume that the failed nodes return in a clean or null state without any knowledge of tracks previously passing through them. We do not consider arbitrary state corruptions in the nodes.

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 two points on the grid; and (3) holes in the network. We discuss these issues in this section.

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 $\sqrt{2}$. If we consider the grid to be of smaller separation than the communication range (denser grid), then the routing stretch factor will decrease, as any straight line will now be approximated more closely when moving along the grid.

4.1 Implementing *Update* on WSN Grid

Storage. For each object P in the network, $trail_P$ is maintained by parent/child pointers at each node in the network. Starting from C, following the child pointers for P will lead to the current location of P. Similarly, starting from p, following the parent pointers at each node will lead to C. Some of the nodes along $trail_p$ are marked as vertex nodes. Each node keeps memory of whether it is a vertex node. Each vertex node at level m keeps memory of c_i for all levels $mx \geq i \geq m$, which is used to determine the smallest level at which an update should start. An array of size $log(N)$ is allocated at each node to store the auxiliary points, where the network is of size $N \times N$.

```
Protocol   Trail at node j in NxN network
Var
        $j.child_p$ : child pointer for object p
        $j.prnt_p$ : parent pointer
        $j.detect_p$ : boolean
        $j.level_p$ : level at which node j belongs
        $j.vertex_p$ : boolean indicating if j is a vertex
        $j.mx_p$ : maximum levels
        $j.c_p$ : array [0..log(N)] of auxiliary points
        $nh$ : auxiliary variable to store the next hop for any message
```

Fig. 8. Trail: state at node j for object P.

The state of node j for a given object P being tracked is shown in Figure 8. The actions involved in updating $trail_P$ (for $b = 1$) are shown in Figure 9. A description of these actions follows.

Actions. We use three types of messages in the update actions. The update actions are initiated at a node where an object P is detected. Recall from Section 2 that this detection is flagged by the underlying detection and association service at the node that is closest to the location of P.

Initially, when an object is detected at a node j, it sends an *explore* message (**Action** $U1$). The parameters of this message are the object id and the location of the object. The explore message travels in around the square perimeters of side lengths, $2^0, 2^1, ...2^{\lfloor log(dist(q,C))\rfloor - 1}$ until it meets $trail_P$, or else the explore message travels to C. Note that if the object is updated continuously as it moves, then the explore message will intersect the trail within a one-hop distance (first level of search). The auxiliary or thought variable *nh* is used to refer to the next hop of any message.

When an explore message is received at a node j (**Action** $U2$), if the node does not belong on $trail_P$, it simply forwards the message to the next hop of exploration. If the explore message intersects $trail_P$, the message is forwarded along its parent pointer until the level m vertex node, where m is the minimal index such that $dist(c_m, p) < 2^{m-1}$ for all i such that $mx \geq i \geq m$. Starting from the level m node where update is started, a new track is created by sending a grow message towards the current location of P. The parameters of a grow message are the identifier of the object, the current location of the object, the level w of the subsequent vertex, the maximum levels of $trail_P$, and the array containing values of c_i for all levels $mx \geq i \geq m$. Geographic routing is used to route the message towards the current location. If the exploration reaches C, the maximum levels of $trail_P$ are reset and a grow message is sent towards the location of P.

Upon receiving a grow message (**Action** $U3$), node j checks to see if it is a vertex node. The node closest to, but outside or on a circle of radius 2^w around, c_w is marked as N_w. This node copies the values of maximum levels in $trail_p$, the level of node j in $trail_P$, and the values of c_i for all $mx \geq i > w$, from the grow message. Additionally, the value of c_w is set to current location of P.

```
$\langle U_1 \rangle :: ((j.detect_p) \wedge (j.child_p \neq j)) \longrightarrow$
          $j.child_p = j;$
          $nh =$ nexthop of exploration;
          $send_{(j,nh)}\ \ (explore(p, j));$
[]
$\langle U_2 \rangle :: recv_{k,j}(explore(p, cur)) \longrightarrow$
  if  $(j == C)$
    $nh =$ nexthop towards $cur$;
    $j.child_p, j.mx_p = nh, \lceil log(dist(C, cur)) \rceil - 1;$
    $send_{(j,nh)}\ \ (grow(p, cur, j.mx_p, j.mx_p, j.c_p));$
  else
    if  $(\neg j.child)$
      $nh =$ nexthop of exploration;
      $send_{(j,nh)}\ \ (explore(p, cur));$
    else
      if  $((j.vertex_p) \wedge (\forall s : (j.mx_p \geq s \geq j.level_p) : (dist(j.c_p[s], cur) < 2^{s-1})))$
        $send_{(j,j.child_p)}\ clear(p)$
        $nh =$ nexthop towards $cur$;
        $send_{(j,nh)}\ grow(p, cur, j.level_p - 1, j.mx_p, j.c_p);$
        $j.child = nh;$
      else
        $send_{(j,j.prnt_p)}\ \ (explore(p, cur));$
      fi
    fi
  fi
[]
$\langle U_3 \rangle :: recv_{k,j}(grow(p, cur, w, x, c)) \longrightarrow$
    if  $(j == cur)$
      $j.prnt_p, j.level_p, j.mx_p = k, w, x;$
    else
      $nh =$ nexthop towards $cur$;
      if  $(dist(j, cur) \geq 2^w)) \wedge (dist(j, nh) < 2^w))$
        $j.vertex_p, j.level_p, j.child_p, j.mx_p = true, w, nh, x;$
        Reset$j.c_p; j.c_p = c; j.c_p[j.level_p] = m;$
        $send_{(j,nh)}\ grow(p, cur, w - 1, j.mx_p, j.c_p);$
      else
        $j.prnt_p, j.level_p = k, w;$
        $send_{(j,nh)}\ grow(p, cur, w, x, c);$
        $j.child_p = nh;$
      fi
    fi
[]
$\langle U_4 \rangle :: recv_{k,j}(clear(p)) \longrightarrow$
        if  $((j.child_p \neq j))$
          $send_{(j,j.child_p)}\ \ (clear(p);$
        fi
        $j.child_p, j.prnt_p, j.vertex_p, j.c_p, j.level_p = \perp, \perp, \perp, \perp, \perp;$
```

Fig. 9. Trail: update actions.

The child and parent pointers are updated to the sender of the grow message and the next hop towards the current location of P, respectively. The grow message is then forwarded to the next hop using geographic routing [Karp and Kung 2000]. If node j is not a vertex node, it simply marks its level in $trail_P$, updates the parent and child pointers, and forwards the message to the next hop towards the current location of P. This procedure is then repeated at subsequent nodes and the trail is updated. If j is the current location of the object, the grow message is not propagated further. Figure 7(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 m node where update is started, a *clear* message is used to delete the old path. Upon receiving a clear message (**Action** $U4$), node j forwards the message to its child and the state of j with respect to object P is reset. The clear message is passed along child pointers of $trail_P$ until the node where object P previously resided.

4.2 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 characterize the upper bound on the find cost in the following lemma, whose proof can be found in Appendix B.

LEMMA 4.1. *The upper bound on the cost of finding an object P at point p from object Q at point q is $(32+3*sec(\alpha)*sec(\frac{\alpha}{2})*\sqrt{2})*d_f$, where d_f is $dist(p,q)$ and $\alpha = arcsin(\frac{1}{2^b})$.*

The actions for the find algorithm are shown in Figure 10. Upon receiving a *find(p,q)* message at node j (**Action** $F1$), if node j does not belong to $trail_P$, then the message is forwarded to the next hop of exploration or else the message is forwarded to the child node. If the current location of P is reached, then a *found* message is sent towards q. Upon receiving a found message (**Action** $F2$), the state of object P is returned to q using geographic routing.

4.3 Fault Tolerance

Due to energy depletion and faults, some nodes may fail, leading to holes in the WSN. Trail supports a graceful degradation in performance in the presence of node failures. As the number of failures increases, there is only a proportional increase in find and update costs as the tracking data structure and the find path get distorted. This is especially good when there are a large number of small holes in the network that are uniformly distributed across the network, as has been the case in our experiments with large-scale wireless sensor networks [Bapat et al. 2005]. We discuss the robustness of Trail under three scenarios: during update, during find and maintaining an existing trail.

```
⟨F_1⟩ :: $recv_{k,j}(find(p,q))$ ⟶
                **if** $(j.child_p == \perp)$
                   $nh$ = nexthop of exploration;
                   $send_{j,nh}$  $(find(p,q))$ ;
                []
                $(j.child_p \neq j) \wedge (j.child_p \neq \perp)$
                   $send_{j,j.child_p}$  $(find(p,q))$ ;
                []
                $(j.child_p = j)$;
                   $nh$ = nexthop towards $p$;
                   $send_{j,nh}$  $(found(p,q))$ ;
                **fi**
[]
⟨F_2⟩ :: $recv_{k,j}(found(p,q))$; ⟶
                **if** $(j \neq q)$
                   $nh$ = nexthop towards $p$;
                   $send_{j,nh}$  $(found(p,q))$;
                **fi**
```

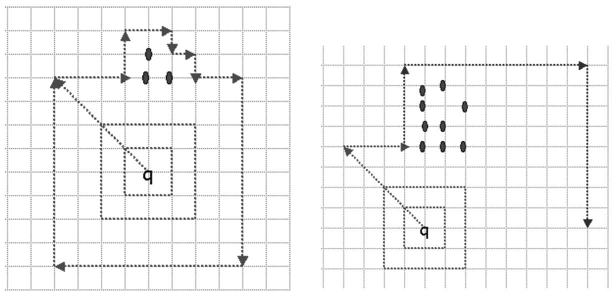
Fig. 10. Trail: find actions.

Tolerating node failures during update. A grow message is used to update a trail starting at a level k node and is directed towards the center of circle $k-1$. In the presence of holes, we use a righthand rule, such as in Karp and Kung [2000], in order to route around the hole and reach the destination. As indicated in the update algorithm for the 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.)

Tolerating failures during a find. We now describe how the find message explores in squares of increasing levels see (Figure 11). When a find message comes across a hole, it is rerouted around the hole using geographic routing only radially outwards of the current level square. If during the reroute we reach a distance from the source of the find corresponding to the next level of search, we continue the search at the next level, and so on. 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.

Maintaining an existing trail. Nodes may fail after a trail has been created. In order to stabilize from these states, we use periodic *heartbeat* actions along the trail. We assume that if a node has a transient failure, the node returns in a null state. We do not handle arbitrary state corruptions in the nodes.

The heartbeat actions are sent by each node along $trail_P$ to its child. At any node r, if a heartbeat is not received from its parent, a search message is sent using geographic routing tracing the boundary of the hole. $trail_P$ is reinforced starting from the first node where the search message intersects $trail_P$ using a *reinforce* message along the reverse path. (If the goal is to find $trail_P$ in the



(a) Find reroute along same level (b) Find reroute to the next level

Fig. 11. Tolerating failures during find.

shortest time, the search should likely be enforced in both directions along the boundary of the hole.)

We have formally stated these maintenance actions in guarded command notation. For reasons of space, we have moved these to the technical report [Kulathumani et al. 2006].

Tolerating failure of C. The terminating point C provides a sense of direction to the trail and serves as a final landmark for the find operation. If C fails, the node that is closest to C will takeover the role of C. However, even in the transient stage when there is no C, the failure of C is tolerated locally. We describe this next.

Consider that C and all nodes in a contiguous region around C have failed. In this case, a search message will be initiated from the node closest to C that belongs to $trail_P$. Because a contiguous set of nodes surrounding C have failed, the search message eventually returns to the node initiating the search by following the boundary of the hole. Thus an existing $trail_P$ terminates at the node that belongs to $trail_p$ and is closest to C. Thus if a find message is unable to reach C, then routing along the boundary of the hole will intersect $trail_P$.

When a new node takes over the role of C, we do not assume that the state of the original C is transferred. The new C has no knowledge of tracks passing through it. Eventually, the maintenance actions for Trail will result in all tracks terminating at C.

5. REFINEMENTS OF TRAIL

In Sections 3 and 4, we have described the basic Trail protocol. 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.

In the first refinement we alter the parameter b to values greater than 1. By increasing b, we update the track of an object more often. This results in straighter tracks with smaller stretch factor. As tracks get straighter, the find at higher levels of the search can follow a triangular pattern (as illustrated in Figure 13), as opposed to complete circles. In fact, as b increases circular explorations can be avoided at more levels of the find. Thus the average find cost in the network decreases as b increases. In sum, by increasing b, we increase the cost of update and decrease the cost of find. This refinement can be used when the rate of updates is small as compared to the rate of find. For example, as the speed of objects in the network decreases, we can increase the value of b.

In the second refinement we change the length of each segment in Trail. In the basic protocol, the segment at each level is a straight line to the next lower level. In this refinement, we modify the length of each segment to be a straight line to the next level plus an arc of length $x \times 2^k$. As x increases, the amount of update at each level increases, but the find exploration can now be smaller. Specifically when $x = 2 \times \pi$, the find is a straight line towards the center of the network. We call this particular parameterization the find-centric Trail protocol.

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 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.

5.1.1 *Optimization of Find.* We now describe the details of this optimization.

LEMMA 5.1. *Given $trail_P$, $(\angle C, p, N_k) < (mx - k + 1) * (arcsin(\frac{1}{2^b}))$, where $(mx \geq k \geq 1)$.* □

The proof can be found in Appendix C.

From here on, we let d_{pC} denote the distance of any object P from C. After the value of mx is defined for a $trail_P$, object P can move for a certain distance before mx is redefined. Therefore, given d_{pC}, the value of mx in $trail_P$ cannot be uniquely determined; however, we note that the value of mx can be bounded, given d_{pC}, and we define $\hat{mx}_p$ as the highest possible value of mx in $trail_P$, given d_{pC}. We now determine $\hat{mx}_p$.

Let R denote the network *radius*, defined as the maximum distance from C. Recall that mx denotes the number of levels in the track for an object P. mx is defined as $\lceil (log(dist(C, p_o))) \rceil - 1$, where p_o is the position of the object when $trail_P$ was (re)created from C. Given network radius R, let $\top$ be the highest number of levels possible for any object in the network. Thus in a given network, $\top = \lceil (log(R)) \rceil - 1$.

LEMMA 5.2. *Given d_{pC}, $\hat{mx}_p = minimum(\lceil log(d_{pC}) \rceil, \top)$.* □

The proof can be found in Appendix D.

Since, given d_{pC}, the index of highest level mx in $trail_p$ cannot be uniquely determined, we state the maximum angle formed by $\angle CpN_k$ in terms of $\hat{mx}_p$ rather than mx. When the actual mx in $trail_P$ is lesser than $\hat{mx}_p$, then the actual maximum angles formed by $\angle C, p, N_k$, where $1 \leq k \leq mx$, is lower than

Exploration level k	Length of triangle base	Height of triangle
$m\hat{x}_q - 1$	$2 * 2^k$	2^k
$m\hat{x}_q - 2$	$2.5 * 2^k$	2^k
$m\hat{x}_q - 3$	$3.1 * 2^k$	2^k
$m\hat{x}_q - 4$	$3.7 * 2^k$	2^k
$m\hat{x}_q - 5$	$4.3 * 2^k$	2^k
$m\hat{x}_q - 6$	$5 * 2^k$	2^k
$m\hat{x}_q - 7$	$6.2 * 2^k$	2^k

Fig. 12. Optimized *find*: pattern of exploration.

the maximum angles stated in the following equation.

$$(\angle C, p, N_k) < (m\hat{x}_p - k + 1) * \left(arcsin\left(\frac{1}{2^b}\right)\right) \tag{15}$$

In the analysis that follows, we characterize the minimum size of exploration required at each level of exploration, given the distance of finder object q from C. Only for ease of explanation do we assume that $b = 3$.

Let Q be the finder at distance d_{qC} from C. Thus $m\hat{x}_q = minimum(\lceil log(d_{qC})\rceil, \top)$. Let P be an object which should be found at the level k exploration. At level k of the exploration, $trail_P$ for any location of P within the circle of radius 2^k around q should be intersected. A circular exploration of radius 2^k around q is sufficient to achieve this. We now characterize the necessary exploration.

We show that, at levels of exploration k where $k \geq m\hat{x}_q - 7$, circular explorations can be avoided and instead a pattern of exploration along the base of an isosceles triangle with apex q and length of base determined by Figure 12 is sufficient to intersect the trails of all objects at distance 2^k from Q. The base of the isosceles triangle is such that $segment(C, q)$ is the perpendicular and equal bisector of the base of the triangle. At levels of exploration $k < m\hat{x}_q - 7$, exploration along the entire circle is necessary. For reasons of exposition, we have moved the procedure to determine the necessary exploration at each level to Appendix E.

Algorithm. Optimized find algorithm (b=3)

(1) Explore at levels k ranging from 0 to $(\lfloor log(d_{qC})\rfloor - 1)$. If $k < (m\hat{x}_q - 7)$, explore using a circle of radius 2^k around q. Else explore along the base of an isosceles triangle with apex q and length of base determined by Figure 12. The base of the triangle is such that $segment(C, q)$ is the perpendicular and equal bisector of the base of the triangle.

An example for the modified find algorithm is shown in Figure 13. In this figure, the object q is at distance 48 units from C. $m\hat{x}_q$ is 6. $\lfloor log(d_{qC})\rfloor - 1 = 4$. Therefore, the levels of exploration are in the range 0..4. Exploration is along the base of triangles at all levels. (This figure is not to scale but for illustration.)

Impact of the optimization. The optimization of find at higher levels is thus significant in that it yields: (1) smaller upper bounds for objects that are far

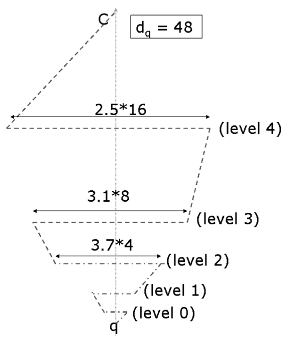


Fig. 13. Optimized find: example.

away from the finder; and (2) lower average cost of $find(p, q)$ over all possible locations of q and p.

As described earlier when $b = 3$, circular explorations are avoided at the highest 7 levels of the find operation. As the value of b increases, the number of levels at which circular explorations can be avoided increases. But by increasing b, we update the track of an object more often. Thus, by increasing b, we increase the cost of update and decrease the cost of find.

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 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

The 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 $Seg(N_k, N_{k-1})$. We call this variation of Trail the find-centric Trail.

5.2.1 *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 those of find queries in a system. Let $trail_P$ for object P consist of segments connecting $C, N_{mx}, .., N_1, p$ 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 $mx \geq k > 0$.

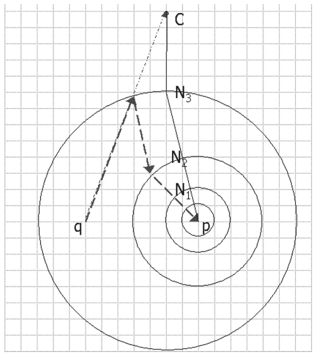


Fig. 14. Find-centric trail.

Starting at q, the find path now is a straight line towards the center, as seen in Figure 14. 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. We characterize the upper bound on the find cost in the following lemma, whose proof can be found in Appendix F.

LEMMA 5.3. *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* $16 * d_f$*, where* $d_f = dist(p, q)$*.*

We note that when events are static, the optimal publish structure is much smaller than publishing along circular tracks. We have studied optimal publish structures for querying in a static context in a related work [Demirbas et al. 2006].

Summary. In this section, we presented two refinements of Trail that lead to a family of protocols. In the first refinement we altered the rate of updates. In the second refinement, we altered the amount of updates at each level. We can choose an appropriate parameterization depending on the expected rate of updates and finds in the network.

As an example, given the expected rate of updates and expected rate of find operations in the network, we can use the value of b in refinement 1 that minimizes the sum of update costs and find cost over a given interval of time. We can compare this cost with that of find-centric Trail and then choose the most appropriate parameterization. The find-centric version of Trail is especially beneficial when the rate of updates is much smaller than that of finds.

6. PERFORMANCE EVALUATION

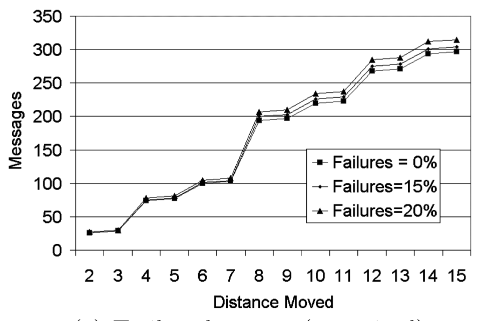
In this section, we first evaluate the performance of Trail using simulations in JProwler [Vanderbilt University 2009] and then describe an experimental evaluation of Trail on a network of 105 Mica2 motes. The goals of our simulation are: (1) to study the effect of routing stretch and discretization errors on the trail stretch factor; (2) to study the effect of uniform node failures on the performance of Trail; (3) to compare the average costs for find and update, as opposed to the upper bounds we derived earlier; and (4) to analyze the performance of Trail when scaled in the number of objects. Our simulation involves a network of 8100 Mica2 motes arranged in a 90×90 grid. We also experimentally evaluate the performance of Trail on a network of 105 Mica2 motes in the Kansei testbed [Arora et al. 2006]. This implementation has been used to support a distributed intruder-interceptor tracking application where the goal of the interceptor is to catch the intruders as far away from an asset as possible. The goals of the experimental evaluation are to study the performance of Trail on a real network and analyze the performance 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.

6.1 Simulation

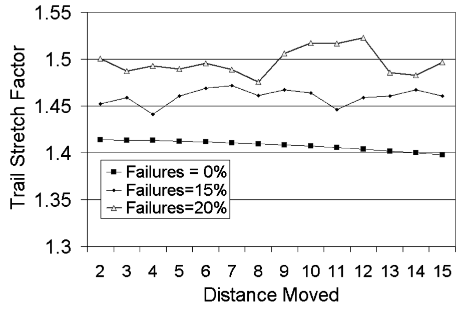
Our simulation involves a 90×90 Mica2 mote network arranged on a grid. The center of the network is placed at one corner, thus essentially simulating one quadrant of the network. This setup lets us test the protocol over larger distances without update and find operations reaching the center. We use JProwler as our simulation platform with a Gaussian radio fading model. The mean interference range is a grid area of 5×5 square units. Packet transmission time is set at 40 ms. We implement geographic routing on a grid to route messages in the network. In the presence of failures we use a lefthand rule to route around the failure [Karp and Kung 2000]. We assume an underlying link maintenance layer because of which the list of *up* neighbors is known at each node.

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 Figure15(a). The jumps visible at distances 4 and 8 show the impact of the $log(d)$ 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 up to 20. We see from the figure that even with failures, the average communication cost increases log linearly with distance. 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, the length of $trail_P$ is less than $1.2 * d_{pC}$. We now study the effect of routing overhead and the discretization factor on the length of the tracking structure that is created. We measure the trail length in terms of number of hops along the structure. Figure 15(b) shows the average ratio of distance from C to the length of the trail during updates over

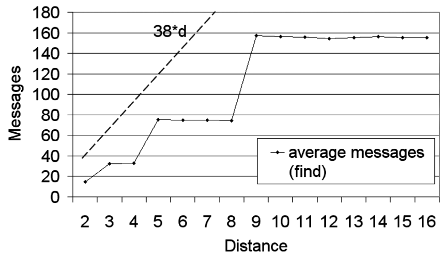


(a) Trail update cost (amortized)

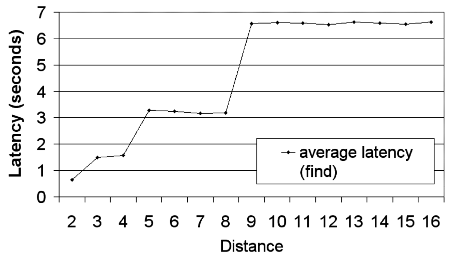


(b) Trail stretch factor

Fig. 15. Trail update costs and trail stretch factor.



(a) average messages (find)



(b) average latency (find)

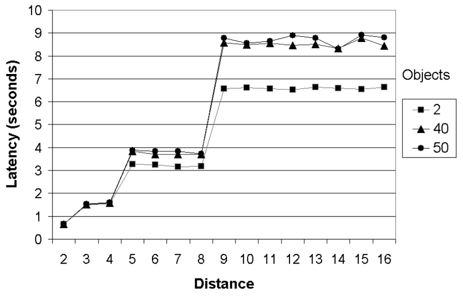
Fig. 16. Trail: find cost.

different distances from the original position. The parameter $b = 1$ in these simulations.

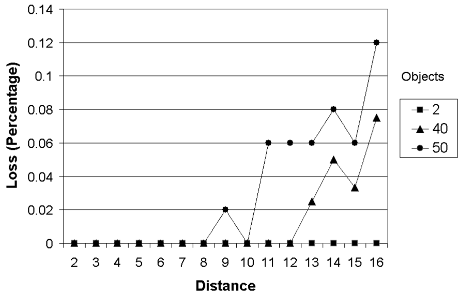
When the trail is first created, the trail stretch factor is equal to the routing stretch factor from C to the original location. In the absence of failures, we notice that the trail stretch factor is around 1.4 at updates of smaller distances and then starts decreasing. This can be explained by the 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 first compare the average find costs of Trail with the upper bounds derived. We study this in the presence of no interference, that is, there is only one finder in the network. 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 $b = 1$. In Figure 16(a) and Figure 16(b), we show the average number of messages and the average latency for the find operation, respectively.

The analytical upper bound $38 * d$ (obtained from Lemma 4.1 for $b = 1$) is indicated using dotted lines in Figure 16(a), and we see that the number of messages exchanged during find operations is significantly lower. When there is only one finder in the network, there is no interference. Therefore, there are no retransmissions except for the case when there is a loss due to probabilistic



(a) average latency (find)



(b) loss percentage (find)

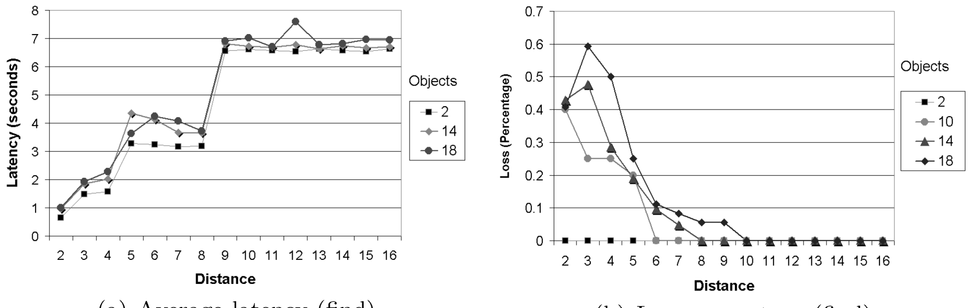
Fig. 17. Effect of interference on find cost.

fading. Therefore the latencies are roughly equal to the number of messages times the message transmission time per hop. The jumps at distances 3, 5, and 9 are due to increase in levels of exploration at these distances. The results of the aforementioned simulations thus validate our theoretical bounds derived in Sections 3 and 4.

Impact of interference. We now evaluate the effect of interference when multiple objects are present in the network. Note that Trail operates in a model where queries are generated in an asynchronous manner. We first evaluate the effect on find latency when objects are uniformly distributed across the network. We then evaluate the performance in a more severe environment where all the objects being found are collocated in the network.

In the presence of interference, messages are likely to be lost and we have implemented the following reliability mechanism to counter this. Forwarding a find message by the next hop is used as an implicit acknowledgment. The find messages are retransmitted up to 4 times by each node. The interval to wait for an acknowledgment is doubled after every retransmission, starting with 100 ms for the first retransmission. Note that 100 ms is a little more than twice the transmission time for each message. Also, upon sensing traffic, the MAC layer randomly backs off the transmission within a window of 10 ms. The maximum number of MAC retries is set to 3.

In the first scenario, we uniformly distribute the objects in a 50×50 area in the center of the quadrant being simulated. By distributing the objects in the middle of the quadrant being simulated, we avoid the decrease in find messages simply because an object is close to the boundary. We simulate 2, 10, 20, 30, 40, and 50 objects in the network. We observe no significant increase in latency from 2 objects up to 30 objects. For the case of 40 and 50 objects in the network, we observe an increase in average latency, especially at larger distances. At larger distances, find messages from different objects interfere to a significant extent. Despite messages being retransmitted up to 4 times, we also see losses during the find operation at distances greater than 12 units. The latency and loss percentages at different distances for 2, 40, and 50 objects are shown in Figure 17(a) and Figure 17(b), respectively.



(a) Average latency (find)

(b) Loss percentage (find)

Fig. 18. Effect of interference on find cost: objects being found collocated.

We now consider a more severe scenario where all the objects being found are at the same location. We compute the average latency for the find operation when objects issuing find query are uniformly distributed around this location, at different distances. Figure 18(a) shows the latency with respect to distances for 2, 14, and 18 objects when all objects being found are at the same location. Figure 18(b) shows the loss percentages for 2, 10, 14, and 18 objects when all objects being found are at the same location. As expected, we see from Figure 18(a) and Figure 18(b) that interference is severe at smaller distances. We see loss percentages as high as 60% when there are 18 objects at small distances.

Summary of evaluation. We observe from the preceding figures that Trail has a find time that grows linearly with distance. When scaled in the number of objects up to 50, with objects uniformly distributed in a 50 $\times$ 50 area and concurrently issuing queries, the query response time still does not increase substantially. However, at a scale of 40 objects and distances of greater than 12 units we observe losses of around 10% during the find operation. This is because at larger distances the find messages from different objects interfere to a significant extent. In a potentially more severe scenario where all objects being found are at the same location and objects issuing find are distributed uniformly around that location, interference is significant at smaller distances. We see loss percentages as high as 60% when there are 9 pairs of objects at small distances.

We now describe an experimental evaluation of Trail on a network of Mica2 nodes, summarize our observations and inferences, and then discuss techniques to handle the effect of interference.

6.2 Experimental Evaluation on Real Network

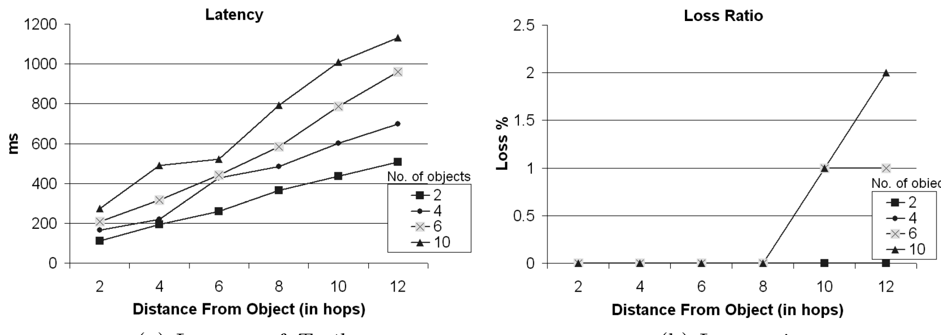
We have implemented Trail for the special case of a long linear topology network for demonstrating an intruder-interceptor tracking application. We have experimentally validated the performance of this version of Trail on 105 Mica2 motes in the Kansei testbed [Arora et al. 2006] under different scaling factors, such as the number of objects in the system, the frequency of queries, and the speed of the objects in the network. This implementation has been used to support a distributed-intruder interceptor tracking application where the goal of the interceptor is to catch the intruders as far away from an asset as possible. This application was demonstrated at Richmond Field Station in Berkeley in 2005 as part of the DARPA NEST Program. In this section, we describe the results of these experiments.

Experimental setup. We use a network of 105 XSM-Stargate pairs in a 15×7 grid topology with 3 ft. spacing in the Kansei testbed. The XSMs comprise a Mica2 family of nodes with the same Chipcon radio but with additional sensors mounted on the sensor board. The XSMs are connected to Stargate via a serial port and the Stargates are connected via Ethernet in a star topology to a central PC. We are able to adjust the communication range by adjusting the power level, and the XSMs can communicate reliably up to 6 ft. at the lowest power level but the interference range could be higher. Trail operates asynchronously with no scheduling to prevent collisions. Hence, we implement an implicit acknowledgment mechanism at the communication layer for per-hop reliability. The forwarding of a message acts as acknowledgment for the sender. If an acknowledgment is not received, then messages are retransmitted up to 3 times.

Object traces. We now describe how the object motion traces are obtained. Motes were deployed in a grid topology with 10 m spacing at Richmond Field Station. Sensor traces were collected for objects moving through this network at different orientations. Based on these traces, tracks for the objects are formed using a technique described in Arora et al. [2004]. These tracks are of the form (timestamp, location) on a 140m $\times$ 60m network. These object tracks are then converted to tuples of the form (id, timestamp, location, grid position), where grid position is the node closest to the actual location on the 15×7 network and id is a unique identifier for each object. These detections are injected into the XSM in the testbed corresponding to the grid position via the Stargate at the appropriate time, using the injector framework in Kansei. Thus, using real object traces collected from the field and using the injector framework, we emulate the object detection and association layer to evaluate the performance of our network service.

Parameters. We evaluate the performance of Trail under different scaling factors such as increasing number of objects, higher speed of objects, and higher query frequency in terms of the reliability and latency of the service. We run Trail with 2, 4, 6, and 10 mobile objects, in pairs. One object in each pair is the object issuing the find query and the other object is the object being located. In each of these scenarios, we consider query frequency of 1 Hz, 0.5 Hz, 0.33 Hz, and 0.25 Hz. The object speed affects the operation of Trail in terms of the rate at which grow and clear messages are generated. We consider 3 different object update rates: one in which objects generate an update every 1 s, every 2 s. and every 3 s. Considering that the object traces were collected with humans walking across the network acting as objects with average speed of about 1 m/s, object update rates of 1 Hz and 0.5 Hz enable a tracking accuracy of 1m and 2m, respectively.

In the 4, 6, and 10 objects scenarios, we consider a likely worst-case distribution of the objects where all objects issuing find and all objects being found



(a) Latency of *Trail* (b) Loss ratio

Fig. 19. Scaling in number of objects (query frequency 0.33 Hz, object update 0.5 Hz).

are within communication range. Moreover, as optimal pursuit control requirements suggest [Cao et al. 2006], the query frequencies depend on relative locations and are lesser when objects are far apart, but we consider all objects issuing queries at the same frequency. If the replies are not received before the query period elapses, then the message is considered lost. The loss percentages are based on 100 find queries at every distance and the latencies are averaged over that many readings.

Scaling in number of objects. Figure 19 shows the latency and loss for find operations as the number of objects increases with query frequency fixed at 0.33 Hz and object updates fixed at 0.5 Hz. Figure 20 shows the latency and loss for find operations as the number of objects increases, with query frequency fixed at 0.5 Hz and object updates fixed at 0.5 Hz.

Scaling in query frequency. Here we analyze how the latency and reliability of Trail are affected as the query frequency increases. In Figure 21, we show the reliability and latency of Trail with 6 objects under query frequencies of 1 Hz, 0.5 Hz, 0.33 Hz, and 0.25 Hz, with object update rate of 0.5 Hz.

Scaling in object speed. Figure 22 shows the latency and loss for find operations with increasing object speeds that generate updates at 0.33 Hz, 0.5 Hz, and 1 Hz. The query frequency is 0.5 Hz and the number of objects is 6.

Summary of experimental evaluation. We observe from the aforesaid figures that Trail offers a query response time that grows linearly with the distance from an object. Trail operates in an environment where objects, can generate updates and queries asynchronously and in such an environment, interference increases the response time. From our experiments, we observe that query latency and loss percentages increase with number of objects and speed of objects, but the loss ratio is not severe. As seen in Figure 19, scaling the number of objects up to 10 yields a loss rate of up to 7% with a query frequency 0.5 Hz and an object update rate of 0.5 Hz. As seen in Figure 22, scaling the object speeds to generating 1 updates results in a loss rate of up to 7%, even with 6 objects in the system and query frequency of 0.5 Hz. Increasing query frequencies has a more severe impact on loss percentages, especially with more objects in the

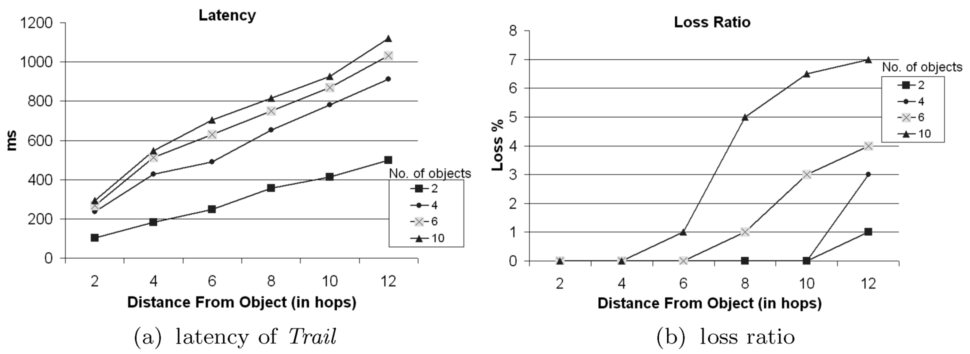


Fig. 20. Scaling in number of objects (query frequency 0.5 Hz, object update 0.5 Hz).

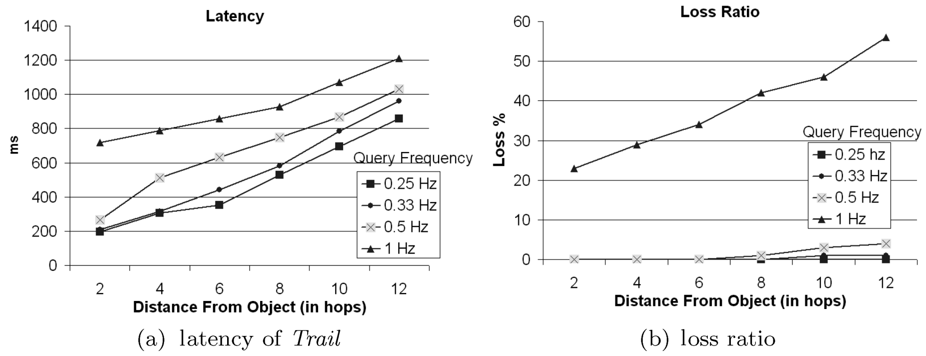
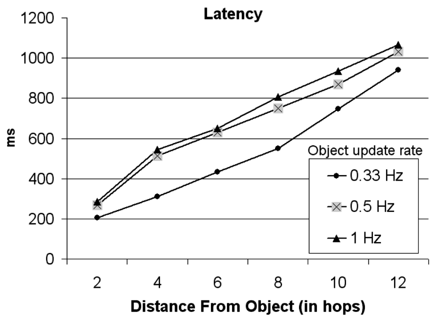


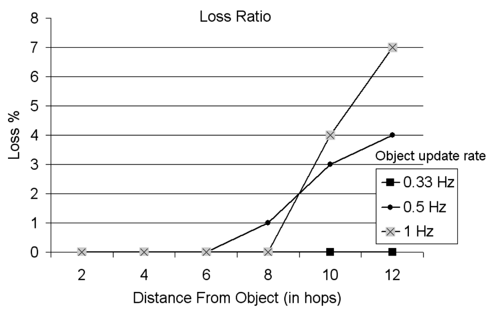
Fig. 21. Scaling in query frequency (6 objects, object update rate 0.5 Hz).

network. In Figure 21, we notice that with 6 objects in the network, loss increases substantially as the query frequency becomes 1 Hz; this happens due to higher interference leading to congestion.

Handling interference. To tolerate network interference, spatial and temporal correlations that exist in the application can be exploited in the following way. The rate at which information is needed by the pursuer is known. The network service can be notified of the next instant at which the state of the evader is needed and the location where the query results need to be sent. The query results can then arrive *just in time*. Advance knowledge about the query and the deadline can be used to decrease the interference in the network when multiple pursuers query about evaders in the network and more so when the objects are densely located. Another possibility to deal with interference is a synchronous *push* version of the network tracking service, where snapshots of objects are published to subscribers in a distance-sensitive manner, thereby avoiding interference. By the same token, applications should be aware of other extreme conditions (in terms of object number and speed) for effectively using the service. For example, applications may compensate for losses by increasing their query frequency, but this should account for extreme scenarios where the increased frequency itself results in higher interference.



(a) latency of *Trail*



(b) loss ratio

Fig. 22. Scaling in object speed (6 objects, query frequency 0.5 Hz).

7. DISCUSSION

In our solution, we made some design decisions like choosing a single point to terminate tracks from all points in the network and avoiding hierarchy in maintaining the tracks. In this section, we analyze these aspects of our solution and compare them with other possible approaches. We find that by avoiding hierarchy, we do not need to partition the network into clusters and maintain these clusters; we can be more locally fault tolerant and we can obtain tighter tracks for any object. We also formally define the notion of terminating points, differentiate these from clusterheads of a hierarchy, and analyze the effect of more terminating points on the maximum find cost and maximum update cost in the network.

Terminating points vs. clusterheads. There are some hierarchy-based solutions [Demirbas et al. 2004; Funke et al. 2006] for the problem of object tracking in a distance-sensitive manner, where the network is hierarchically partitioned into clusters and information of objects is maintained at clusterheads[1] at each level. Even in these solutions, information about an object is published across the network to local clusterhead(s) at each level in the hierarchy, all the way up to one or more clusterheads at the highest level in the hierarchy. We call these points at the highest level of the hierarchy as terminating points.

Formally, a terminating set τ is a smallest set of points such that tracks of objects from every location in the network pass through at least one point in τ. The cardinality of a set τ is denoted as μ_τ. There can be one or more terminating sets, each with one or more terminating points. In Stalk [Demirbas et al. 2004] there is a unique clusterhead at the highest level, thus there is a single terminating set with a single terminating point. In LLS [Abraham et al. 2004] and DSIB [Funke et al. 2006], there are multiple clusterheads at the highest level. Thus there are multiple terminating sets, each with one terminating point. Tracks from every point in the network pass through each of these clusterheads. Thus, each clusterhead at the highest level constitutes

[1]Note that the responsibility of a clusterhead for every cluster could be shared by multiple nodes and not necessarily just one node. We refer to the abstraction of a leader for every cluster as clusterhead.

a terminating set by itself. In Trail there is a unique terminating set with a unique terminating point, namely C.

It is in the process of maintaining tracks from a terminating point that we have avoided hierarchy in Trail. In hierarchy-based solutions, to maintain tracks and to answer queries, tracks from terminating points necessarily pass through these clusterheads, whereas Trail avoids hierarchy by determining anchors for the tracking paths on-the-fly based on the motion of objects.

Merits of avoiding hierarchy. By avoiding hierarchical solutions we do not need either a distributed clustering service that partitions the network into clusterheads at different levels and maintains this clustering or a special (maybe centralized) allocation of infrastructure nodes. By avoiding a hierarchy of such special nodes Trail is also more locally fault tolerant. For example, in the case of a find operation, failure to retrieve information from an information server at a given level would require the find to proceed to a server at the higher level [Funke et al. 2006]. This is particularly expensive at higher levels of the hierarchy. On the other hand, in Trail a find operation redirects around a hole created by failed nodes using routing techniques such as the lefthand rule [Karp and Kung 2000] and such faults can be handled, in a sense, proportional to the size of the fault. Similarly, tracks to existing objects can be repaired more efficiently. As the number of failures increase, there is only a proportional increases in find and update costs.

Moreover, avoiding hierarchy allows for minimizing the length of tracking paths, given a terminating point. We analytically compare the performance of Trail with that of other hierarchy-based solutions for tracking objects in Section 8 and we observe that Trail is more efficient than other solutions. Trail has about 7 times lower update 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.

Choice of terminating points. In the technical report Kulathumani et al. [2006], we have formally analyzed the choice of a unique terminating point for tracks from all points in the network and the trade-offs associated with multiple terminating sets and multiple terminating points per set in terms of the maximum find and update costs in the network. We provide a summary of our analysis here.

We show in our analysis that in order for find to be distance sensitive, it must the case that all terminating points in a terminating set be traversable in $O(N)$ where the network is of $N \times N$ dimensions. This precludes the possibility of a track from every point terminating at itself (i.e., the terminating set being equal to the set of all points in the network). So the question arises as to what are the choices for the number of terminating points and terminating sets. We consider three cases: a single terminating set with multiple terminating points, multiple terminating sets, each with one terminating point; and a different terminating point for each *type* of object.

Single Terminating Set with Multiple Terminating Points. Intuitively, there exists a possibility of decreasing the maximum track length in the network by dividing the network into regions and having a local terminating point per region. The maximum track length and therefore the maximum update cost in the network thus depends on the size of the largest region. We are faced with the question of how small can these regions be? We show that in order to maintain find distance sensitivity, the diameter of the largest region can only be a constant order less than the diameter of the network, namely, at least $\Omega(N)$ in an $N \times N$ network. Thus, there can be only an constant order of cost decrease compared to having only one terminating point. Moreover, decreasing the maximum update cost by dividing the network into smaller regions results in a proportionate increase in the maximum find cost. This is because if from any finder location, a track belonging to an object in any location in the network is to be found, then it must be the case that the find trajectory contains all points in the terminating set, thereby increasing the worst-case find cost.

Multiple Terminating Sets Each with One Terminating Point. If there are multiple terminating sets then it is sufficient for find to traverse the terminating point in any such set. In this case there is a likelihood of decreasing the maximum find cost when compared to having only set of terminating points because tracks can be found by reaching a terminating point in any of the terminating sets. The maximum find cost in the network depends on the size of the largest region. However, we show in Appendix A that to maintain update distance sensitivity, the size of the largest region has to be at least $\Omega(N)$ in an $N \times N$ network. Also when the number of terminating sets is greater than 1, update has to traverse the terminating point in all terminating sets in the worst case. Thus the maximum update cost in the network increases.

Maintaining a track with respect to local terminating points could be advantageous if it is more likely that the querying object and the object being found are closer. Thus, a find will never run into the scenario of having to traverse all regions in the network. Similarly, maintaining a track with respect to multiple terminating point sets could be advantageous if objects are likely to move within bounded regions within a network. In this article we consider all distances between querying object and tracked object to be equally likely and do not restrict mobility of the objects. Hence, we consider only the case where there is a unique terminating set with a single terminating point, namely C.

Different Terminating Point for Each Type of Object. We note that it is also feasible to select a different terminating point for different *types* of objects. In this article we describe how to maintain tracks for objects with respect to one terminating point and to guarantee find and update distance sensitivity. A different terminating point can be chosen for each type of object based on hash functions and each type of finder object can choose the respective terminating points as a worst-case landmark. Nevertheless, this concept is orthogonal to that of maintaining tracks with respect to a given terminating point and it is therefore compatible with Trail.

We now summarize some of the performance aspects of Trail in terms of load balancing (fairness), maintenance, and memory.

Load Balancing and Fairness. In hierarchy-based solutions, to maintain tracks and to answer queries, tracks from terminating points necessarily pass through these clusterheads, whereas Trail avoids hierarchy by determining anchors for the tracking paths on-the-fly based on the motion of objects. By avoiding hierarchy, in Trail load is in fact more evenly distributed (more fairness) than hierarchical solutions in which queries have to answered by specific locations and the same nodes are taxed.

Maintenance. The tracks maintained in Trail are almost straight with a stretch factor of less than 1.2. By maintaining shorter tracks that are not convoluted, the cost of maintaining the tracks in Trail is actually lower. We have also discussed in detail the fault-tolerance actions of Trail in Section 4, where we describe how to handle failures of nodes along existing tracks, failures during update and find and also how failures of terminating point C can be handled locally.

Tolerating Failure of the Terminating Point. In Section 4 we have shown that in Trail, we can locally tolerate the failure of even C. Our protocol actions are such that when C fails or a set of nodes in a contiguous region around C fail, the track for an object will terminate at any node that is closest to the boundary formed by the hole. Thus during a find operation, a redirection rule as in GPSR is bound to intersect the track and once again the faults are tolerated locally.

Memory efficiency. We note that the tracks maintained in Trail only contain pointers to the current location of an object and not the state information of the object. We also note that the storage required at each node is $O(log(N))$ per object, where the network is of size $N \times N$.

Handling bottleneck at C. We have previously discussed the option of having multiple terminating sets, each with one terminating point. One example of this is to construct a circle of constant radius δ around the center of the network and to define each point on this circle as a terminating set. Thus tracks from any point in the network pass through all points on this circle. A find trajectory can intersect any point on this circle to obtain a pointer to the object. The maximum update cost increases by a factor of δ and the maximum find cost decreases by a factor of δ, while still maintaining distance sensitivity. The responsibility of handling queries is now distributed evenly around C.

8. 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 [Awerbuch and Peleg 1995; Demirbas et al. 2004; He et al. 2006]

	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. 23. Trail: analytical comparison.

and we have focused our attention on WSN support for tracking. Some network tracking services [Dolev et al. 1995] 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 Awerbuch and Peleg [1995], Demirbas et al. [2004], and Abraham et al. [2004] that provide distance-sensitive updates and location; see Figure 23.

Locality Aware Location Services (LLS) [Abraham et al. 2004] is a distance-sensitive location service designed for mobile ad hoc 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 $128 * d_m * log(d_m)$, where d_m is the distance an object moves, as opposed to the $14 * d_m * log(d_m)$ 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 [Demirbas et al. 2004] 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 $96 * d * log(d)$. This increase occurs because of having to query neighbors at increasing levels of the partition in order to establish lateral links for distance sensitivity [Demirbas et al. 2004].

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 Awerbuch and Peleg [1995], a hierarchy of regional directories is constructed and the communication cost of a find for an object d_f away is $O(d_f * log(N))$ and that of a move of distance d_m is $O(d_m * log(D) * log(N))$ (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 nonlocal clustering program that constructs sparse covers.

Querying and storage. Querying for events of interest in WSNs has also received significant attention [Intanogonwiwat et al. 2003; Ratnasamy et al. 2002; Liu et al. 2004] and some of that research focuses on distance-sensitive querying. We note that Trail, specifically the find-centric approach, can also be used in the context of static events.

In Liu et al. [2004], 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 Liu et al. [2004]. In directed diffusion [Intanogonwiwat et al. 2003], 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. Rumor routing [Braginsky and Estrin 2002] 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.

Geographic Hash tables [Ratnasamy et al. 2002] 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 multilevel 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.

Distance-Sensitive Information Brokerage [Funke et al. 2006] protocol performs a hierarchical clustering of the network and information about an event is published to neighboring clusters at each level. Using find-centric Trail we can query information about static events in a distance-sensitive manner. We also note that when events are static, the optimal publish structure is much smaller than publishing along circular tracks. We have studied optimal publish structures for querying in a static context in a related work [Demirbas et al. 2006].

Spatio-temporal query services. Motivated by a class of applications in which mobile users are interested in continuously gathering information in real time from their vicinities, a network data service called *spatio-temporal query* has been proposed in Lu et al. [2005]. The spatial constraint for the network service comes from an energy-efficiency point of view; only nodes relevant to a query area should be involved in contributing the query result. The temporal constraint is due to a requirement on data freshness for the query results. An approximate motion model for the mobile user is assumed. Specifically, the motion can be predicted over a small interval of time. The query area at any time is a function of the current location of the mobile user. The key difference in Trail is that the query area in consideration is the entire network, as opposed to a function of the querier location as in Lu et al. [2005].

We note that when Trail is used in the context of a distributed pursuer evader game, spatial and temporal correlations that exist in the application can be exploited using ideas presented in Lu et al. [2005] to improve the performance of the application and the network. The rate at which information is needed by the pursuer is known. The network service can be notified of the next instant at which the state of the evader is needed and the location where the query results need to be sent. The query results can then arrive *just in time*. Constraints on evader speed can be exploited as follows. Subsequent queries for evader location can originate from the previous evader location and the results can be routed back to the pursuer.

9. 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 update costs at almost equal find costs and can tolerate faults more locally as well.

Importantly, Trail maintains tracks from object locations to only one terminating point, the center of the network. 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 is also able to decrease the upper bound find costs at larger distances and thereby decrease the average find cost across the network.

Trail is a family of protocols and 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 90×90 network using JProwler. We have also successfully implemented and tested the 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. This is especially true 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.

We also note that spatial and temporal correlations that exist in the application can be exploited to improve the performance of the application and the network. Two such examples are as follows. (1) The rate at which information is needed by the pursuer is known. The network service can be notified of the next instant at which the state of the evader is needed and the location where the query results need to be sent. The query results can then arrive *just in time*. Advance knowledge about the query and the deadline can be used to decrease the interference in the network when multiple pursuers query about evaders in the network, and more so when the objects are densely located. (2) Constraints on evader speed can be exploited as follows. Subsequent queries for evader location can originate from the previous evader location and the results can be routed back to the pursuer. Thus the energy efficiency of the network can be improved.

REFERENCES

ABRAHAM, I., DOLEV, D., AND MALKHI, D. 2004. LLS: A locality aware location service for mobile ad-hoc networks. In *Proceedings of the Joint Workshop on Foundations of Mobile Computing (DIALM-POMC)*. ACM, 75–84.

ARORA, A., DUTTA, P., BAPAT, S., KULATHUMANI, V., ZHANG, H., NAIK, V., MITTAL, V., CAO, H., DEMIRBAS, M., GOUDA, M., CHOI, Y., HERMAN, T., KULKARNI, S., ARUMUGAM, U., NESTERENKO, M., VORA, A., AND MIYASHITA, M. 2004. a line in the sand: a wireless sensor network for target detection, classification, and tracking. *Comput. Netw. Special Issue on Military Communications Systems and Technologies 46,* 5, 605–634.

ARORA, A., ERTIN, E., RAMNATH, R., NESTERENKO, M., AND LEAL, W. 2006. Kansei: A high fidelity sensing testbed. *IEEE Internet Comput. 10,* 2, 35–47.

ARORA, A. AND RAMNATH, R. 2004. ExScal: Elements of an extreme wireless sensor network. In *Proceedings of the 11th International Conference on Embedded and Real-Time Computing Systems and Applications*, 102–108.

AWERBUCH, B. AND PELEG, D. 1995. Online tracking of mobile users. *J. Assoc. Comput. Mach. 42*, 1021–1058.

BAPAT, S., KULATHUMANI, V., AND ARORA, A. 2005. Analyzing the yield of ExScal, a large scale wireless sensor network experiment. In *13th IEEE International Conference on Network Protocols*.

BRAGINSKY, D. AND ESTRIN, D. 2002. Rumor routing algorithm for sensor networks. In *Proceedings of the 1st ACM Workshop on Wireless Sensor Networks and Applications*. ACM, 22–31.

CAO, H., ERTIN, E., KULATHUMANI, V., SRIDHARAN, M., AND ARORA, A. 2006. Differential games in large scale sensor actuator networks. In *Information Processing in Sensor Networks (IPSN)*. ACM, 77–84.

DEMIRBAS, M., ARORA, A., AND KULATHUMANI, V. 2006. Glance: A light weight querying service for sensor networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*. 242–257.

DEMIRBAS, M., ARORA, A., NOLTE, T., AND LYNCH, N. 2004. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, 143–162.

DOLEV, S., PRADHAN, D., AND WELCH, J. 1995. Modified tree structure for location management in mobile environments. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 530–537.

FUNKE, S., GUIBAS, L., NGUYEN, A., AND WANG, Y. 2006. Distance sensitive information brokerage in sensor networks. In *Proceedings of the International Conference on Distrbuted Computing in Sensor Systems (DCOSS)*. Springer, 234–251.

HE, T., KRISHNAMURTHY, S., AND STANKOVIC, J. 2006. VigilNet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sensor Netw. 2,* 1, 1–38.

INTANOGONWIWAT, C., GOVINDAN, R., ESTRIN, D., HEIDEMANN, J., AND SILVA, F. 2003. Directed diffusion for wireless sensor networking. *IEEE Trans. Netw. 11,* 1, 2–16.

KARP, B. AND KUNG, H. T. 2000. Greedy perimeter stateless routing for wireless networks. In *Proceedings of Mobile Computing and Networking, (MobiCom)*. 243–254.

KULATHUMANI, V., ARORA, A., DEMIRBAS, M., AND SRIDHARAN, M. 2006. Trail: A distance sensitive wsn service for distributed object tracking. Tech. rep. OSU-CISRC-7/06-TR67, The Ohio State University.

Liu, X., Huang, Q., and Zhang, Y. 2004. Combs, needles, maystacks: Balancing push and pull for discovery in large-scale sensor networks. In *Proceedings of the ACM Conference On Embedded Networked Sensor Systems (Sensys)*. ACM, 122–133.

Lu, C., Xing, G., Chipara, O., Fok, C.-L., and Bhattacharya, S. 2005. A spatiotemporal query service for mobile users in sensor networks. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, 381–390.

Ratnasamy, S., Karp, B., Yin, L., Yu, F., Estrin, D., Govindan, R., and Shenker, S. 2002. GHT: A geographic hash table for data-centric storage. In *Proceedings of the 1st ACM Workshop on Wireless Sensor Networks and Applications (WSNA)*. ACM, 12–21.

Sarkar, R., Zhu, X., and Gao, J. 2006. Double rulings for information brokerage in sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCOM)*. ACM, 286–297.

Shin, J., Guibas, L., and Zhao, F. 2003. A distributed algorithm for managing multi-target indentities in wireless ad hoc networks. In *Proceedings of the International Workshop on Information processing in Sensor Networks (IPSN)*. 223–238.

Sinopoli, B., Sharp, C., Schenato, L., Schaffert, S., and Sastry, S. 2003. Distributed control applications within sensor networks. *Proc. IEEE. 91*, 1235–46.

Vanderbilt University. JProwler, discrete event simulator for wireless networks. `http://www.isis.vanderbilt.edu/Projects/nest/jprowler/index.html`.

Received May 2007; revised April 2008; accepted May 2008