Trunk and Trail: Wireless Sensor Network Services For Distributed Object Tracking

Vinodkrishnan Kulathumani[1], Mukundan Sridharan[1], Murat Demirbas[2],
Hui Cao[1], Emre Ertin[1], and Anish Arora[1]

[1]Dept. of Computer Science and Engineering, The Ohio State University
[2]Dept. of Computer Science and Engineering, State University of New York at Buffalo

Abstract

In-network observation and control of mobile objects via static wireless sensors demands consistent, timely information. In this paper, we present two wireless network (WSN) services that support such observation and control: Trunk *provides a global snapshot of the locations of all mobile objects at well-known times to all in-network subscribers, and* Trail *provides the location of a particular mobile object upon demand to an in-network subscriber. We design both services to be energy efficient, reliable, and fault-tolerant despite the network dynamics typically associated with WSNs. Because* Trunk *service times are well-known and subscribers demand consistent information,* Trunk *operates readily in a synchronous model and achieves energy-efficiency by scheduling radio transmissions and receptions. By way of contrast, as* Trail *service times are unknown and subscribers demand timely information,* Trail *basically operates in an asynchronous model and achieves low latency via a distributed data structure that is updated only locally when objects move and offers a find time that increases linearly with the distance from an object. In the asynchronous model however, concurrent "where" operations can interfere with each other and receivers are always on, so we consider an extension to* Trail *for the synchronous model, which yields better reliability and energy efficiency but at the cost of latency. We provide experimental results on the performance of these services in a network of 105 motes (specifically XSMs) in our* Kansei *WSN testbed and describe an example object tracking system that we demonstrated based on these services which involved an intruder-interceptor game for perimeter protection WSNs.*

Keywords: Wireless sensor networks, tracking, distributed control, real-time snapshots, distance sensitivity, stabilization, energy efficiency

1 Introduction

Tracking of mobile objects has received significant attention in the context of military applications, mobile computing and cellular telephony. In some systems developed thus far, tracking is performed by an ad-hoc network of mobile nodes; in others, it is performed by a network of static nodes. In this paper, we focus on the latter system model and, in particular, consider a wireless sensor network (WSN). We moreover focus our study on network services that support tracking-based applications. These include applications that monitor objects, as in the *ALineInTheSand* [2, 18], *Vigilnet* [9], and *ExScal* [1, 17] WSN systems, but also applications that "close the loop" by performing tracking-based control; examples include pursuer-evader tracking, as in PEG [13] where a controller's objective would be to minimize the average catch time of all evaders, and intruder-interceptor tracking applications [4], where a controller's objective would be to maximize the average catch distance of all intruders from some "goal line".

Most previous work in tracking-based control applications [21] has been centralized or semi-centralized. That is, the controller has not been a distributed one. Our study is motivated by the consideration of large-scale WSN deployments such as *ExScal*. As a result, the study of distributed controllers becomes desirable, especially for reducing the latency with which controller components receive their relevant state information. And in turn controller distribution raises issues in the consistency, reliability, fault-tolerance, and energy-efficiency of network services that support the required track information collection.

Requirements of network services for tracking-based control: We illustrate these requirements by way of an example. In *ExScal*, a large number of wireless sensor nodes are deployed in a long, linear perimeter that runs along a long valuable asset (the "goal" line). Intruders enter the perimeter with the intention of crossing over to the goal line. The basic task of the network is to detect, classify and track these intruders. In the control scenario we consider, this information is used to enable interceptors within the perimeter to "catch" the intruders as far from the goal line as possible. This scenario can be realized by providing the controller with (i) global knowledge of the system state to assign intruders to interceptors and (ii) the location/tracking information about their respective intruder for the interceptors to move.

The first control function, namely assignment control, can be distributed by embedding a controller in each interceptor and providing consistent global state information of the system (i.e., about all intruders and interceptors) to all interceptors. This suggests a push model for an underlying network service. Consistency of this information requires avoiding the case where objects move between local snapshots and are thus missed/duplicated in the collected state; achieving consistency is simplified if there is pre-planning of the times at which all nodes take their local snapshots.

The second control function, namely motion control, is readily distributed on a per interceptor basis (and is simplified if there is at most one interceptor assigned to each intruder). We have elsewhere [4] shown Nash equilibrium conditions for versions of the intruder-interceptor game, hereby referred to as *IIG*. Our results imply that, for satisfying optimality constraints, the rate at which an interceptor requires information about the intruder it is tracking is not constant and it depends on the relative locations of the two: the closer the distance the faster the rate. This suggests a pull model for an underlying network tracking service that supports at an interceptor, the location tracking of its assigned intruder. It also suggests a *distance sensitivity* requirement on the service, whereby the latency involved in location decreases as the objects get closer to each other.

Other requirements of the network services are generic to WSN dynamics. Given the complex channel characteristics and the occurrence of collisions, the reliability of the network services –measured in terms of information loss– should be high. Given that network nodes may fail or their state may be corrupted, the services should be appropriately tolerant to these faults. And since sensor nodes are energy-constrained devices, where even idling/listening on the radio is a significant power draw as compared to transmission, the services must be energy-efficient.

Summary of the paper: In this paper, we present two latency-efficient, energy-efficient, reliable, and fault-tolerant network services that support tracking-based distributed control applications. We first describe an architecture for distributed, mobile object tracking systems using wireless sensor networks. We then design the two services, which we call *Trunk* and *Trail*: the specification for *Trunk* is to return a consistent global snapshot

of the state of all objects in the system to all subscribers at regular intervals. Since service times are well-known, *Trunk* operates in a synchronous, push model. By being synchronous, nodes in the network listen on the radio or transmit only when scheduled thus enabling energy efficiency.

The specification for *Trail* is to return the location of a particular object in response to an in-network controller issuing a *where* query regarding that object. To this end, *Trail* maintains a tracking data structure by propagating mobile object information, by which it satisfies a distance sensitivity property. The time taken to complete the *where* operations is linearly proportional to the distance between the objects. Also, when objects move, the time taken and work performed to update the tracking structure is linearly proportional to the distance moved. By operating in a local manner, *Trail* achieves energy efficiency. However, *Trail* operates in an asynchronous model where the applications can invoke a *where* operation at any time and nodes are always *awake* to listen on the radio. Since this is energy consuming, we design a *Synchronous Trail* service in which maintaining the tracking data structure and the *where* operations are exactly like Trail, but the nodes in the network operate synchronously.

We have implemented *Trunk* and *Trail* and we provide an experimental validation of the performance of these *Trunk* and *Trail* in a network of 105 motes (of type XSMs that are derivatives of Mica2) using *Kansei*, a wireless sensor network testbed at Ohio State [7]. We note that *Trunk* and *Trail* have been used in a demonstration of the distributed intruder-interceptor game discussed above at Richmond Field Station in Berkeley in 2005 as part of the DARPA NEST program.

Organization of the paper: In Section 2, we describe the control system architecture, model of the network and its faults, and the specification of the *Trunk* and *Trail*. In Sections 3 and 4, we respectively detail *Trunk* and *Trail* and present the results of experimental evaluation of their performance in a wireless sensor network testbed. In Section 5, we describe *Trail* in a synchronous model. In Section 6, we discuss related work and in Section 7, we make concluding remarks and discuss extensions and future work.

2 System Architecture, Model and Specification

In this section we present an architecture for distributed mobile object tracking systems, describe the network and fault model, and formally state the specifications for the network services in the system.

2.1 System Architecture

The system comprises *mobile objects*, and a network of static devices called *motes*. The tracking application runs on a subset of the mobile objects and uses the sensor network to track desired mobile objects. Fig. 1 shows the overall architecture of the system and how the components are interconnected.

Each mote consists of a sensing component and a radio component. Each mote also participates in three low level network services namely, object detection and association, clustering and time synchronization and participates in two types of tracking services, a snapshot service and an object location service. Associated with each mobile object is an *agent*, which resides on one or more motes in the network. Agents are responsible for storing the state associated with an object and also act as a communication interface with the network. We now describe the network model.

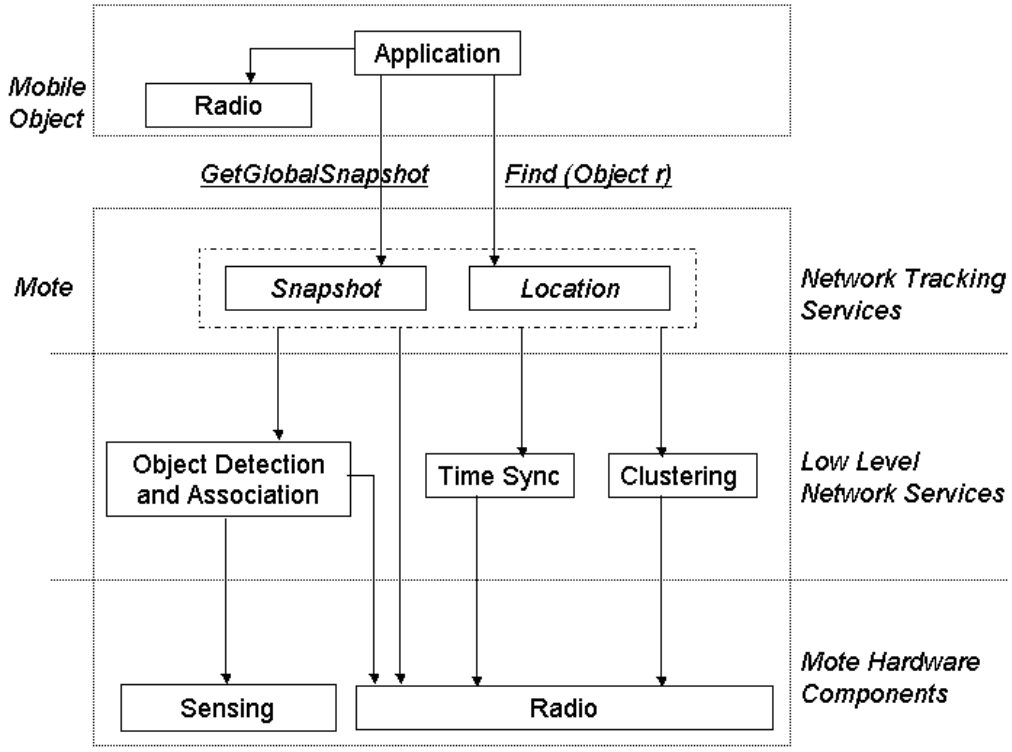


Figure 1. Architecture of Mobile Object Tracking System

Network Model: The motes in the network are partitioned into L clusters with a cluster-head for each cluster. These cluster-heads form the backbone of the network. We present our network services in this paper by assuming that the backbone motes are arranged in a linear topology. We discuss extensions of our services to 2-d topologies in Section 8. There can be upto n mobile objects in the network.

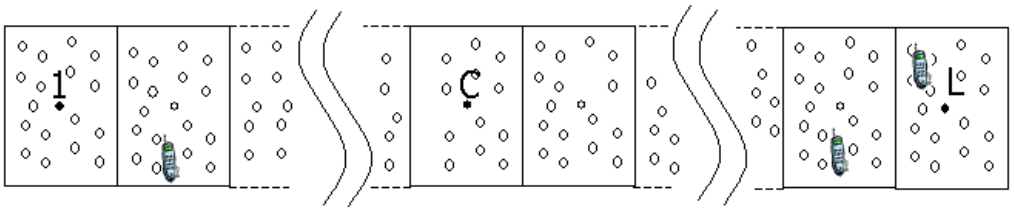


Figure 2. Network of Sensor Nodes and Mobile Objects

Notations: As a convention in this paper, $j.r$ refers to a variable r at mote j. $j.bbid$ refers to the id of the cluster in which mote j belongs to. We denote one cluster in the network as a central cluster and its cluster-head is denoted as C. Variables specific to the network services, *Trunk* and *Trail* are defined in the respective sections. Objects are denoted by their ids obj_p. Also, $obj_p.r$ refers to a variable r at object obj_p.

Fault Model: The network can corrupt or lose a message if it interferes with any other message sent at the same time. Motes in the network can failstop and restart. Motes can suffer arbitrary state corruption.

Starting from an arbitrary state, we show that our services are self-stabilizing to a consistent state.

2.2 Network Services Specification

We now describe the network services that support the system in some detail.

Clustering Service: The clustering service partitions the network into clusters. The specification for this service is that all nodes within a cluster are within communication range of each other and of nodes in the neighboring clusters but not beyond that. Such clustering can be realized using a distributed local clustering protocol such as LOCI [15]. The clusterheads form the *backbone* for the network. The clustering service provides each non-backbone mote with the id of its backbone mote, denoted as $j.bbid$, and the backbone motes with the ids of their neighboring backbone motes. Every mote j has an *in-neighbor* that refers to the neighbor towards the center ($j.unbr$) and an *out-neighbor* that refers to the path away from the center ($j.dnbr$). For all non-backbone motes and backbone motes farthest from the center, $j.dnbr$ is set to $\perp$. For all non-backbone motes, $j.unbr = j.bbid$. Also $C.unbr = \perp$.

Object Detection and Association Service: The object detection service uses the sensing and radio components to compute the location of each object and also corroborates object detections with previous detections for the same object. The service is specified as follows. The object detection service assigns a unique id p in the range $1..n$ to every object in the network. Whenever an object p is detected, it signals an event $\mathbf{detected_p}$ and atomically sets $\mathbf{j.detected_p}$ at the mote j that is closest to the object p. This mote is called as the *agent* for object p. The service also signals a $\mathbf{moved_p}$ event and atomically resets $\mathbf{j.detected_p}$ if object p moves and j is no longer the agent for p. The invariant maintained by this service is stated below.

- I0: $j.detected_p$ is set iff j is an agent for object p
- I1: If $j.detected_p$, then j is the mote closest to the object p

Time Synchronization Service: This service establishes a notion of global time across the motes in the network and is used by services that operate in a synchronous model. Time synchronization may be implemented in a number of ways. The nodes could be GPS enabled or the nodes could run a periodic distributed beaconing protocol [8]. The invariant for this service is that for any two motes j and k in the network, $j.t = k.t$.

Network Tracking Services: These provide snapshot services that return the state of a set of objects or location services to which, informally the specification could be to get the location of a set of objects or the nearest object that satisfies certain conditions. In this paper we present two such network tracking services for mobile objects, namely *Trunk* and *Trail*.

The specification for *Trunk* is to return a consistent global snapshot of the state of all mobile objects. *Trunk* answers queries by the application of the form $\mathbf{GetGlobalSnapshot(T, n)}$ where T is the period at which the snapshots are needed and n is the number of mobile objects in the system. Each querying application gets an identical snapshot of the system.

Trail offers the following function: $\mathbf{where(obj_i, obj_p)}$. $\mathbf{where(obj_i, obj_p)}$ returns the state of the object i, including its location at the current location of the object p issuing the query. *Trail* offers a distance sensitivity property. The time taken to complete the *where* operations and the amount of work for this operation in terms of the number of messages exchanged is linear with respect to the distance between the objects. Also the time

and work to update the tracking structure is proportional to the distance moved.

Communication Abstraction: Communication between motes i and j are achieved by using $\mathbf{send_{i,j}}$ $(\mathbf{m})$ and $\mathbf{receive_{i,j}}$ $(\mathbf{m})$, where i is the sender and j is the receiver. Motes i and j could be multi-hop. Note that $dist(x, y)$ is defined as the hop distance between x and y. Thus two backbone motes in clusters r apart have hop distance $2 + r$. $\mathbf{send_i}$ $(\mathbf{m})$ is used by mote i to send a message to all nodes within 1-hop of i. Similarly $\mathbf{receive_j}$ $(\mathbf{m})$ can be used to receive messages sent within 1 hop of j. δ is assumed to be the 1-hop message transmission time.

3 Trunk

Trunk is a snapshot service that disseminates a consistent global snapshot of the system containing the state of all mobile objects to all subscriber objects. The specification for the functionality of *Trunk* is of the form $\mathbf{GetGlobalSnapshot(T, n)}$, where T is the interval at which the snapshots are required and n is the number of objects in the system. *Trunk* exploits the knowledge of the period T and schedules the collection of individual object snapshots and distribution of the global snapshot.

3.1 Description

Let the network have $L = 2l + 1$ clusters with l clusters on either side of the central backbone mote C. Recall that for any mote j, $j.unbr$ denotes the neighboring backbone mote towards the center and $j.dnbr$ denotes the neighboring backbone mote away from the center. For non-backbone motes, $j.unbr$ is the nearest backbone mote. And that if and only if a mote j is an agent for object i, a $\mathbf{detected_i}$ event is raised and $j.detected_i$ is set.

The state of object i at any time t is denoted as $state_i(t)$. Node j's view of the state of all objects is given by $j.snapshot$. $j.snapshot$ is the union of individual object snapshots. j's view of snapshot for object i is denoted as $j.snapshot_i$. The timestamp of the snapshot is given by $j.snapshot_i(ts)$. Every subscribed object gets the global snapshot once at interval boundaries sT where s is an integer, from its agent mote at that instant.

Each snapshot consists of two phases: During the update period, individual object updates are sent from the non backbone motes to the nearest backbone mote. During the wave period, the individual snapshots from backbone motes are gathered and the global snapshot is formed at C and dispersed to the subscribed mobile objects. We assume the period T between snapshot is at least long enough to accommodate an update period and a wave period.

Update Period: We first describe the actions during an update period. At the beginning of the update period, denoted as *SnapshotTime*, motes that are agents for any object i at that time record the state of that object in $j.snapshot_i$. During the update period, the non-backbone motes within a cluster gather the snapshots within the cluster and send to the backbone nodes. Since *Trunk* has knowledge that there are n objects in the system, n slots are reserved for the updates to be gathered among the non-backbone motes and sent to the backbone node. The duration of each slot equals the per-hop transmission period δ. The mote with the update for the object with highest id in the cluster is responsible for sending the snapshot to the nearest backbone node. We call this mote the *leader* for the cluster for that interval. Thus we ensure that in every period T, there is at most one update message to any backbone node and each backbone mote is thus awake for just one slot during the update period.

The gathered snapshots are sent to the backbone motes in two slots. Since motes within a cluster can be heard at most one cluster away, there is no message interference during this process.

Wave Period: The wave period itself consists of two phases, a snapshot gathering phase and a snapshot dispersing phase. Each backbone mote has one slot to transmit in each of these phases. During the gathering phase, snapshots from individual backbones are gathered starting from the motes farthest from the center and going towards the center. In any slot during this phase, two backbone motes are scheduled except the nodes that are one hop away from C. Thus $l+1$ slots are reserved for the gathering phase. C aggregates all snapshots and initiates a dispersion phase. This message contains the global snapshot for the period. $l+1$ slots are reserved for the dispersion phase as well. All nodes that are agents for subscribed objects, listen to the dispersion message and communicate it to the objects at the end of the last slot.

Note that backbone motes send a message during the snapshot gathering phase only if they hear an update during that period from a non-backbone mote or if they receive a snapshot from its neighboring backbone mote farther from the center. Thus no messages are transmitted if there are no objects in the system. Schedules within a period T are shown in Fig. 3.

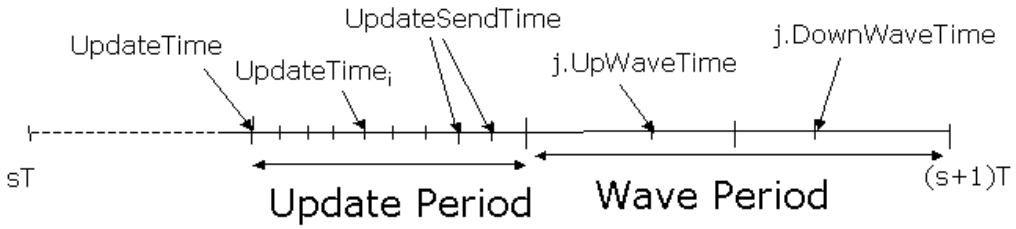


Figure 3. Trunk: Schedule for Motes Within an Interval

Definition 3.1 (SnapshotTime)**.** *SnapshotTime is the time during an interval when the update gathering process starts for the current interval. All nodes that are agents for objects, record the snapshot atomically at this time. SnapshotTime is same across all motes.*

Definition 3.2 (UpdateTime)**.** *$UpdateTime_i$ is the time during an interval in which a non-backbone node j that is an agent for object i, can send $j.snapshot$. $UpdateTime_i$ is same across all motes.*

Definition 3.3 (UpdateSendTime)**.** *$j.UpdateSendTime$ is the time during an interval in which the leader for the cluster sends the aggregated snapshot to its clusterhead. Motes in alternate clusters have the same $UpdateSendTime$.*

Definition 3.4 (UpWaveTime)**.** *$j.UpWaveTime$ is the time during an interval in which a backbone mote j can send its local snapshot to its* in-neighbor*.*

Definition 3.5 (DownWaveTime)**.** *$j.DownWaveTime$ is the time during an interval in which a backbone mote j can send the global snapshot to its* out-neighbor*.*

Definition 3.6 (LastSnapshotTime)**.** LastSnapshotTime(t) *at any time t, is the most recent SnapshotTime.*

Based on the values of T, n, l, δ, each node j calculates $SnapshotTime$, $UpdateTime_i$ for all i in the range $1..n$, $j.UpdateSendTime$, $j.UpwaveTime$ and $j.DownWaveTime$. Resetting the network with changes to any of

these parameters is a global operation and there exist self-stabilizing solutions for the same. The *Trunk* protocol at node j is shown in Fig. 4.

Protocol Trunk at node j
Constant n : number of objects
Var $j.snapshot_i$: state of object i at node j
$j.leader$: boolean
$j.detect_i$: boolean
Actions
$\langle S_1 \rangle :: j.detect_i \wedge ((j.t \ mod \ T) = SnapshotTime) \longrightarrow$
$j.snapshot_i = state_i(j.t)$;
if $(\neg j.bb)$
$j.leader = true$;
$j.maxi = max_i(j.snapshot_i \neq \perp)$
fi;
[]
* * * * * * * * *
Actions at non-backbone nodes
$\langle NBB_1 \rangle :: (j.leader) \wedge ((j.t \ mod \ T) = UpdateTime_{j.maxi}) \longrightarrow$
$send_j$ $(j.snapshot)$;
[]
$\langle NBB_2 \rangle :: (j.leader = true) \wedge recv_k(m) \wedge (j.bbid = k.bbid) \wedge ((j.t \ mod \ T) = UpdateTime_p) \longrightarrow$
if $(j.maxi > p)$
Update $j.snapshot$
[]
$(j.maxi < p)$
$j.leader = false$;
$\forall i :$ **set** $j.snapshot_i = \perp$
fi;
[]
$\langle NBB_3 \rangle :: (j.leader) \wedge ((j.t \ mod \ T) = j.UpdateSendTime) \longrightarrow$
$send_{j,j.unbr}$ $(j.snapshot)$;

$leader = false$;
[]
* * * * * * * * *
Actions at backbone nodes
$\langle BB_1 \rangle :: recv_{j.dnbr,j}(m) \longrightarrow$
Update $j.snapshot$
[]
$\langle BB_2 \rangle :: (\exists i : (j.snapshot_i \neq \perp)) \wedge ((j.t \ mod \ T) = j.UpWaveTime) \longrightarrow$
$send_{j,j.unbr}$ (j.snapshot)
$\forall i$: **set** $j.snapshot_i = \perp$;
[]
$\langle BB_3 \rangle :: recv_{j.unbr,j}(m) \longrightarrow$
Update $j.Snapshot$
[]
$\langle BB_4 \rangle :: (\exists i : (j.snapshot_i \neq \perp)) \wedge ((j.t \ mod \ T) = j.DownWaveTime) \longrightarrow$
$send_{j,j.dnbr}$ $(j.Snapshot)$;
[]
* * * * * * * * *
Reset Snapshot
$\langle RS_1 \rangle :: ((j.t \ mod \ T) = 0) \longrightarrow$
$\forall i$: s.t$(j.detect_i)$ $send_{j,i}$ $(j.Snapshot)$
$\forall i$: **set** $j.snapshot_i = \perp$

Figure 4. Trunk: Network Service for Global Snapshots to Mobile Objects

3.2 Correctness

All motes satisfy their local invariant I at all times t. I contains the following conditions:

- I1: If $(j.snapshot_i \neq \perp)$ then
  $((j.snapshot_i(ts) = LastSnapshotTime(t)) \wedge (j.snapshot_i = state_i(\text{LastSnapshotTime(t)})))$

- I2: If $(((j.t \bmod T) < j.SnapshotTime) \vee ((j.t \bmod T) > 0)))$
  then $j.snapshot_i = \perp$

We now state the global correctness properties for Trunk in terms of the following lemmas. These are established from the program actions and the above local invariants.

Lemma 3.1. *At any time t such that t mod T = SnapshotTime, for every object i there exists only one node j such that* $j.snapshot_i = state_i(t)$.

Proof. Only one node is an agent for an object at any time and the node which is an agent at *SnapshotTime* records the state of that object. □

Lemma 3.2. *At any time t, for all objects i and all motes j,* $(j.snapshot_i = \perp)$ *Or* $(j.snapshot_i(ts) =$ *LastSnapshotTime(t)).*

Proof. Since nodes record the snapshot for the interval at the same time, snapshot for all objects are timestamped with *LastSnapshotTime(t).* □

Hence, we can state:

Lemma 3.3. *At any time, if there exist motes j and k, such that for any object i,* $j.snapshot_i \neq \perp$ *and* $k.snapshot_i \neq \perp$*, then it must be that* $j.snapshot_i = k.snapshot_i$.

Lemma 3.4. *At* $C.DownWaveTime$*, for all motes j where* $j \neq C$, $j.snapshot = \perp$

Proof. After any mote j sends the snapshot to its *in-neighbor* using action $BB2$, $j.snapshot$ is set equal to $\perp$. Thus C is the only mote that initiates the dissemination of the global snapshot. □

Using the above lemmas, local invariants and actions of the backbone motes, we state the following theorem.

Theorem 3.7. *For any two motes k and p, at times sT where s is an integer, k.snapshot is equal to p.snapshot and for all i in the range* $1..n$, $k.snapshot_i$ *equals* $state_i(LastSnapshotTime(sT))$

Thus all objects get a consistent global snapshot of the system in all intervals.

3.3 Performance

In this subsection, we characterize the performance of *Trunk* in terms of energy, latency and reliability. Latency depends on the round trip time across the backbone of the network and the time required to send the updates to the backbone nodes. Latency determines the staleness of the state of every object in the global snapshot that is received by the objects in every interval.

Theorem 3.8. *The state of every object i in the global snapshot received by any object in every period T is stale by* $(L + n + 3)\delta$ *time.*

Proof. In every interval, the agents for all subscribed objects send the global snapshot to the objects at the end of the wave period. In each slot in the wave period, two backbone nodes transmit except at one hop away from the center. The wave period thus takes $(2l + 2)$ slots. Updates for n objects are gathered in n slots and sent to the backbone in 2 slots. Hence the state of every object i in the global snapshot received by an object in every period T is stale by $(L + n + 3)\delta$ time. □

We now characterize the energy efficiency of *Trunk* in terms of the number of messages transmitted by the nodes and the amount of time nodes have to be awake listening on the radio.

Theorem 3.9. *Backbone motes other than center C in* Trunk *listen on radio for at most $(3\delta/T)$ fraction of time and transmit at most 2 messages in every snapshot period T, and C listens on radio for at most $(3\delta/T)$ fraction of time and transmits at most 1 message in every snapshot period T*

Proof. Each backbone mote listens in one slot every snapshot period for update from non-backbone motes in the cluster, listens in one slot during the wave gathering phase for snapshot from its *out-neighbor* and listens in one slot during the dispersion phase for the global snapshot received from its *in-neighbor*. (Note that the center C has two *out-neighbors*). □

Note: Only the non-backbone motes that have $j.detect_i$ set at the *SnapshotTime* in any interval, are *on* during the update period to gather the local snapshots and send to the backbone mote. This way, the backbone motes are awake for only one slot to listen to an update every interval and we also achieve load balancing among the motes in the network.

3.4 Fault Tolerance

We now show that *Trunk* is self stabilizing to its invariant conditions starting from an arbitrary state. Suppose $j.snapshot_i$ is corrupted at any mote j and I is violated, since the snapshots are reset after every interval, *Trunk* stabilizes from these conditions. $j.leader$ is also reset after every interval as seen in action NBB_3.

For message reliability in the network, *Trunk* schedules the transmissions in such a way that there is no interference. To tolerate failures along the backbone node until a lower level service repairs the network, the following scheme could be used. An alternate backbone exists that supervises the regular backbone. An alternate backbone node on any cluster overhears any message for the backbone node on the same cluster. If the regular backbone node does not transmit at the scheduled time, the alternate backbone node transmits in the next slot. This scheme results in increasing the width of a slot to 2δ.

Motes can be added or removed maintaining the cluster properties. However changes in parameters T, n or l have to be communicated to the entire network. The existing backbone itself can be used to disseminate the new query.

3.5 Experimental Evaluation

In this section, we evaluate the performance of *Trunk* experimentally using *Kansei*, a wireless sensor network testbed.

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 are attached to the Stargate via serial port. The XSMs also have a Chipcon radio. 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. The Stargates have a 802.11b wireless card and they are also connected via ethernet in a star topology to a central PC. For convenience, let us number the rows 1..7 and columns 1..15 in the testbed.

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 [2]. 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 inject framework in *Kansei*. This message corresponds to the $detected_i$ event for any object i. Similarly, a message is injected at the XSM corresponding to the previous grid position, corresponding to the $moved_i$ message. 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 services.

Implementation details: We let the motes on row 4 to be the backbone. We evaluate *Trunk* in two different cluster settings. In the first setting, the 105 nodes are divided into five 3×7 clusters of 21 nodes each. In this setting, we can test *Trunk* upto a scale of 5 clusters. Further, in order to test the performance of *Trunk* over larger number of clusters, we vary the number of clusters up to 15 and in all configurations with more than 5 clusters, there are 7 nodes per cluster.

In order to minimize interference, the nodes use varying power levels as described below. During the update gathering phase, the non-backbone nodes use a high enough power level so that all nodes within a cluster can hear each other. In this phase, since the nodes transmit based on the ids of the objects, there is no interference. When sending the updates to the backbone, the non-backbone nodes switch to a lower power level so that they can reach the backbone node. The backbone nodes operate at the lowest power level while transmitting, since they need to reach the adjacent backbone nodes which are 9 ft away in 3×7 cluster setting and 3 ft in the 7 node cluster setting. Even under these reduced power levels, it is likely that there is interference during the update from non-backbone nodes to the backbone or when the waves along the backbone motes approach the center.

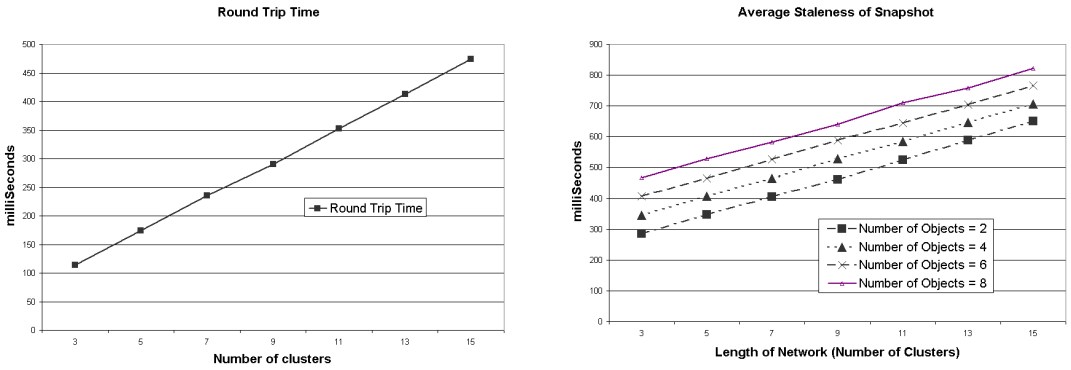
Trunk is implemented in *TinyOS* and downloaded on the XSMs using a programming interface provided by *Kansei*. *Trunk* accepts the length of the network, the snapshot period and the number of objects as parameters. These parameters can be injected using the trace injector framework, thus enabling the evaluation of *Trunk* under different parameters. The location of a mote in terms of its grid position is also injected using the same injector framework. Based on the parameters injected and the per-hop transmission time, *Trunk* calculates the schedules for transmitting, listening and taking a local snapshot. The per-hop transmission time is conservatively chosen to be 30 msec based on packet transmission experiments. Using the locations injected, a mote also determines its *in-neighbor* and *out-neighbor*.

Performance: We now describe the experimental results for *Trunk* in terms of its reliability and latency. The round trip time along the backbone for *Trunk* depends only on the length of the backbone network and not on the number of mobile objects. The round trip time is measured at the farthest backbone node for clusters of different length. Since the nodes schedule their transmissions based on the per-hop transmission time, as shown in the Fig. 5(a), the round-trip time increases linearly with the number of clusters and there is little variance.

The staleness of a global snapshot received by an object depends on the length and the number of objects in the network. This is shown in Fig. 5(b). This is measured by injecting object detection traces obtained from Richmond Field station and recording the global snapshots received at nodes in different clusters along the backbone.

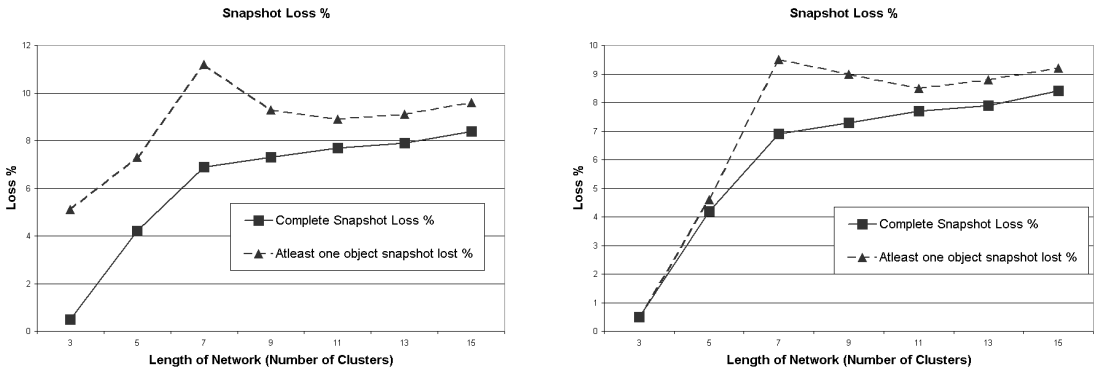
We characterize the reliability of Trunk as follows. Consider an object that is farthest from the center. Either the entire global snapshot could be lost in the network (this implies with high probability that the loss occurred during the wave period as the probability of all 6 updates to the backbone being lost is quite low) or the snapshot for a particular object or set of objects could be lost. The loss ratios as the length of the network increases, are shown in Fig. 6(a). The loss ratios are over 500 snapshot intervals and 6 objects in the network.

Evaluation: The experiments confirm the analysis that the latency (which translates to the staleness of a snapshot) grows linearly with the number of clusters in the network and the number of objects. Losses along the backbone during the wave period grow with the length of the network but the rate of growth decreases after a point indicating that losses occur mainly in the region close to C. Losses during the update period are likely due to interference that occurs when the updates sent from non-backbone nodes to the backbone motes. Note in Fig. 6(a) that the update loss rises to a maximum when number of clusters is 7 and then decreases; this is because in this case the backbone motes are 3 ft apart and given 6 objects, it is more likely that leaders are formed one cluster apart. Since we have only 2 slots to send the updates to the backbone, this can result in interference. By increasing the number of slots for sending updates from non-backbone to backbone nodes to 3, the loss rates are reduced, as is shown in Fig. 6(b).



(a) Round Trip Time (b) Staleness of Snapshots

Figure 5. Average Round Trip Time and Staleness of Snapshots in *Trunk*



(a) Updates to Backbone in 2 Slots (b) Updates to Backbone in 3 Slots

Figure 6. Percentage of Snapshots Lost in *Trunk*

4 Trail

In this section, we describe *Trail*, a network service for tracking mobile objects in a local and distance sensitive manner. *Trail* offers the following interface: $\mathbf{where(obj_i, obj_p)}$, that returns the location of object i at the current location of the object p, issuing the query. To implement this function, *Trail* maintains a tracking data structure by propagating mobile object information obtained through the object detection and association service. We first describe how the tracking data structure is maintained when the objects move.

4.1 Tracking Data Structure

In *Trail*, a path is maintained for every object i in the system from the center C to the current location of the object, using pointers c_i and p_i at each mote. Initially, $j.p_i$ and $j.c_i$ equals $\perp$ for all i and j. When an object i exists in the system, starting from C and following the pointer c_i leads to the agent for object i. p_i is a reverse pointer for object i and starting from the agent for object i and following p_i would lead to C. The paths are established and maintained by means of *grow* and *clear* messages. The protocol for maintaining the tracking data structure is stated in guarded command notation in Fig. 7 and we describe this protocol in brief here.

When a mote becomes an agent for an object, the mote starts forming the path for the object i by sending a *grow* message towards the center through the *in-neighbors* and sets c_i to point towards the object along the path. The *grow* message stops when it reaches the C or if it reaches a mote where c_i is already set. Starting from this mote, a *clear* message is propagated which resets the pointer c_i to the old location of object i. When a $\mathbf{moved_i}$ event is raised at any mote j, j simply sets c_i and p_i to $\perp$.

Correctness: In the absence of faults, every mote j satisfies at all times invariant I comprising the following conditions:

- I1: Iff j is agent for object i, $j.c_i = i$
- I2: If $j.c_i \neq \perp$, then $j.p_i \neq \perp$ or j will send a grow towards center and set $j.p_i$
- I3: If $j.p_i \neq \perp$, then $(j.p_i).c_i = j$ or $j.p_i$ has sent a clear message to j or j has sent a grow towards center

Protocol Trail at node j
Constant $HeartBeatTime$: integer
Var
$j.c_i$: pointer to object i
$j.p_i$: pointer away from object i
$j.sendHbTimeout_i$: integer
$j.receiveHbTimeout_i$: integer
Actions

Path Update Actions

$\langle U_1 \rangle$ :: **detected$_i$** $occurs \longrightarrow$
  if $(\neg(j.c_i = \perp))$
    $send_{j,j.c_i}$ $(clear(i))$;
    $j.c_i = i$;
  []
  $(j.c_i = \perp)$
    $j.c_i = i$;
    if $(\neg j.center)$
      $send_{j,j.unbr}$ $(grow(i)$;
    fi;
    $j.p_i = j.unbr$
  fi;
[]
$\langle U_1 \rangle$ :: **moved$_i$** $occurs \longrightarrow$
  $j.c_i, j.p_i = \perp, \perp$;
[]
$\langle U_2 \rangle :: recv_{k,j}(grow(i)) \longrightarrow$
  if $(\neg(j.c_i = \perp))$
    $send_{j,j.c_i}$ $(clear(i))$;
    $j.c_i = k$;
  []
  $(j.c_i = \perp)$
    $j.c_i = k$;
    $send_{j,j.unbr}$ $(grow(i)$;
    $j.p_i = j.unbr$
[]
$\langle U_3 \rangle :: recv_{k,j}(clear(i)) \longrightarrow$
  if $(\neg j.c_i)$
    $send_{j,j.c_i}$ $(clear(i))$;
    $j.c_i, j.p_i = \perp, \perp$;
  fi;

Object Location Actions

$\langle F_1 \rangle :: recv_{k,j}(where(obj_i, obj_p)) \longrightarrow$
  if $(j.c_i \neq \perp \wedge j.c_i \neq i))$
    $send_{j,j.c_i}$ $(where(obj_i, obj_p))$
  []
  $(j.c_i = i) \wedge (j.c_p \neq \perp)$
    $send_{j,j.c_p}$ $(here(state_i, obj_p))$
  []
  $(j.c_i = i) \wedge (j.c_p = \perp)$
    $send_{j,j.unbr}$ $(here(state_i, obj_p))$
  []
  $(j.c_i = \perp)$
    $send_{j,j.unbr}$ $(where(obj_i, obj_p))$
  fi;
$\langle F_1 \rangle :: recv_{k,j}(here(state_i, p)) \longrightarrow$
  if $(j.c_p \neq \perp \wedge j.c_p \neq p))$
    $send_{j,j.c_p}$ $(here(state_i, p))$
  []
  $(j.c_p = p)$
    $send_{j,p}$ $(here(state_i, p))$
  []
  $(j.c_p = \perp)$
    $send_{j,j.unbr}$ $(here(state_i, p))$
  fi;

Figure 7. Trail: Network Service for Tracking Mobile Objects

Stabilizing Actions for Track Updates

$\langle S_1 \rangle :: (j.p_i \neq \perp) \wedge (j.time - j.SendHbTimeout_i = HeartBeatTime) \longrightarrow$
  $send_{j,j.p}$ $(hearbeat_i)$;
  $j.SendHbTimeout_i = HeartBeatTime$
[]
$\langle S_2 \rangle :: recv_{k,j}(heartbeat_i) \longrightarrow$
  $j.c_i = k$;
  $j.p_i = j.unbr$;
  $j.ReceiveHbTimeout_i = HeartBeatTime$
[]
$\langle S_3 \rangle :: (j.c_i \neq i) \wedge (j.c_i \neq \perp) \wedge (j.time - j.ReceiveHbTimeout_i = HeartBeatTime) \longrightarrow$
  $j.c_i = \perp$;

Figure 8. Trail: Stabilizing Actions

which will be received by $j.p_i$.

- I4: If $j.c_i = \perp$, then $j.p_i = \perp$

A path for an object i, denoted as $trail_i$ is a sequence of motes $(j_1, .., j_x, ..., C)$, such that $j_1.c_i = i$ and j_1 is the agent for object i at that instant and every other node c_i points to a mote that is closer to object i. A consistent state for *Trail* with respect to an object i is one in which $trail_i$ exists and $j.p_i = \perp$ for every mote j not in the sequence.

Following the program actions and invariant condition I1, I2 and I4, we can derive:

Lemma 4.1. *Starting from an initial state, if* $\mathbf{detected_i}$ *occurs at any node j, then* Trail *reaches a consistent state in $dist(j, C) \times \delta$ time.*

Proof. If j is the first agent for object i, the *grow* message starting from j reaches the center in $dist(j, C)$ hops and the path to object i is complete in $dist(j, C) \times \delta$ time. □

Lemma 4.2. *If* $\mathbf{moved_i}$ *occurs at any node j then $j.c_i = \perp$ and eventually there exists no mote k such that $j = k.c_i$.*

Proof. Let p be the mote where $detected_i$ occurs, when $moved_i$ occurs at mote j. Mote p will send a grow message. In $dist(p, j) \times \delta$ time, a clear message will be received at node j. At this time, there will be no process k such that $j = k.c_i$. □

Using the previous lemma, we have:

Lemma 4.3. *Starting from a consistent state in* Trail*, when an object i moves distance d, a consistent state is reached in $dist(p, j) \times \delta$ time where p and j are the new and old agents for object i respectively.*

Stabilization: We now state stabilization actions that re-establish the invariant conditions upon starting from an arbitrary state.

I1 is established trivially by the event $\mathbf{detected_i}$. Conditions I2 and I4 can be re-stablished by local correction. However the system could be in a corrupted state such that for any object i, $trail_i$ is broken or there exist dead paths that do not lead to any object, thus violating $I3$. To stabilize despite this, we use periodic *heartbeat* messages for every object i in the system. Every mote that has a valid p_i sends out a heartbeat message. If a mote j receives a *hearbeat* message from k, then $j.c_i$ is set to k. Thus broken paths are re-established. The periodic **heartbeat** messages also serve to remove dead paths. If a process does not hear a grow message although $j.c_i$ is valid, $j.c_i$ is set to $\perp$. The stabilizing actions are $s1$, $s2$ and $s3$, as shown in Fig. 8.

4.2 Locating an object

We now describe how *Trail* responds to queries of the form $\mathbf{where(obj_i, obj_p)}$, where p is the mobile object that initiated the query. Let $p.agent$ be a mote k. If k itself is not the agent for i, then k sends the query to its

in-neighbor. If the path to i is known at this mote, then the query is forwarded along that path or else the query is sent further towards the center. If the object i exists in the network, then the $\mathbf{where(obj_i, obj_p)}$ message starting from mote k will reach a mote j such that $j.c_i \neq \perp$ in at most $dist\ (k, C)$ hops. If the object i lies on the other side of the center as k, then C will be the first mote such that $c_i \neq \perp$. The object location operation is illustrated in Fig. **??** and the actions in guarded command notation are shown in Fig. 7.

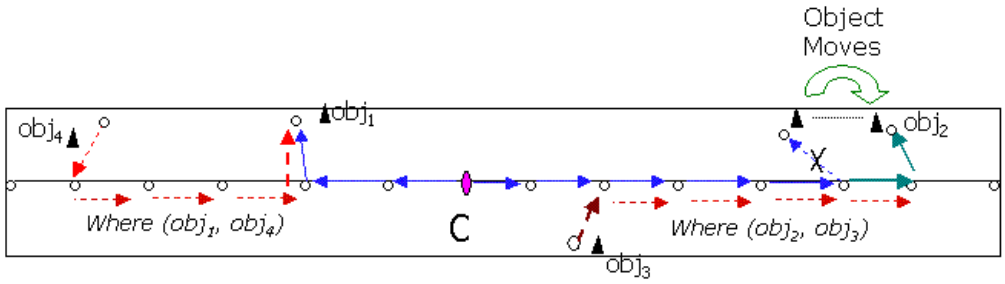


Figure 9. Trail: Illustration of Track Updates and Object Location

The object i is said to be located when the $\mathbf{where(obj_i, obj_p)}$ message reaches a mote r such that $r.c_i = i$.

Theorem 4.1. *When there are no object updates in the system and* Trail *is in a consistent state, when an object p issues a query* $\mathbf{where(obj_i, obj_p)}$*, the object i is located in at most s hops where $s = dist(k.agent, i.agent)$*

Once the object is located, the reply to $\mathbf{where(obj_i, obj_p)}$ message is propagated through the $\mathbf{here(state_i, obj_p)}$ message along the path to object p. The $\mathbf{here(state_i, obj_p)}$ message reaches $p.agent$ exactly like how the $\mathbf{where(obj_i, obj_p)}$ message reaches $i.agent$. Thus we have:

Theorem 4.2. *The latency between an object p issuing a* $\mathbf{where(obj_i, obj_p)}$ *to* Trail *and receiving a* $\mathbf{here(state_i, obj_p)}$ *in* Trail *increases linearly with the distance between objects i and p.*

Proof. For any two objects p and i, recall that $dist(i.agent, p.agent)$ is proportional to the physical distance d between the two objects. The query from object p reaches $i.agent$ in $dist(i.agent, p.agent)$ hops and the reply reaches $p.agent$ in $dist(i.agent, p.agent)$. □

Stabilization: We have shown in Section 4.1 that the tracking data structure is self-stabilizing from an arbitrary state. Here, we discuss stabilizing actions during object location. If a *where* or *here* message is lost either when *Trail* is in an inconsistent state or due to message losses, the stabilizing action for locating object i is implemented using a timeout at $p.agent$, the agent of the object p sending the query. After the timeout, the agent re-issues the query. The timeout is chosen according to the network diameter and δ, the per hop transmission time. Note that if object k moves, the state of k is transferred to the new agent and hence the timeout value as well.

Note: In case $trail_i$ is being updated during object location, the $\mathbf{where(obj_i, obj_p)}$ message can reach a process r such that $r.c_i = \perp$, which reflects a previous location of the object i. In this case, the return value can be the earlier location or $\perp$ depending upon the application requirement.

4.3 Experimental Evaluation

In this section, we describe the performance of *Trail* using experiments conducted in Kansei. The experimental setup in the testbed is as described in Section 3. The clusters are of size 7. The backbone motes operate at a power level by which they can reach 3 ft reliably while the non-backbone nodes operate at a power level sufficient to reach the backbone node. Note that even at these power levels, there can be interference with other messages, and *Trail* operates asynchronously with no scheduling to prevent collisions. Hence, we implement an implicit acknowledgement 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.

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 always in pairs. One object in each pair is the object issuing *where* query and the other object is the object being located. In each of this scenario, 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 second, every 2 seconds and every 3 seconds. 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. Note that each update can generate multiple *grow* and *clear* messages.

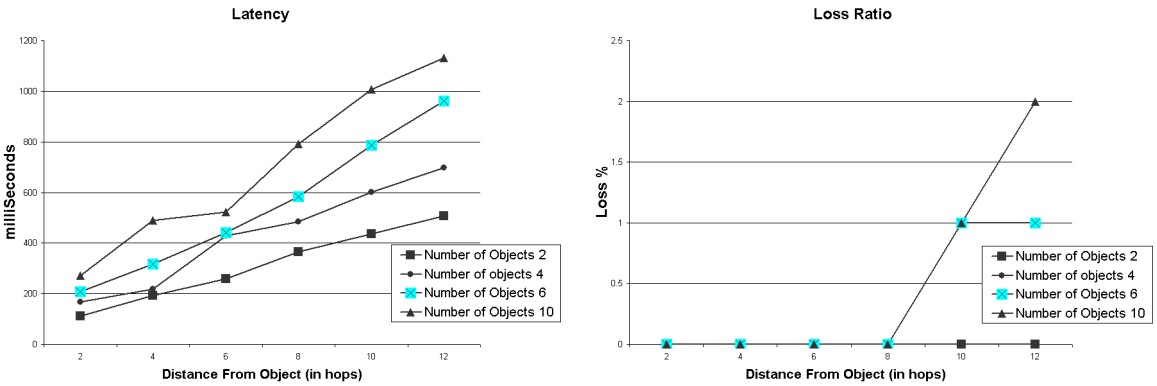
In the 4, 6 and 10 objects scenario, we consider a likely worst case distribution of the objects where all trackers are in the same cluster and all objects being found are also in the same cluster. Moreover, as IIG application requirements suggest [4], the query frequencies depend on relative locations 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 elapsed, then the message is considered lost. The loss percentages are based on 100 *where* queries at every distance and the latencies are averaged over that many readings.

Scaling in number of objects: Fig. 10 shows the latency and loss for *where* operations as the number of objects increases with query frequency fixed at 0.33 Hz and object updates fixed at 0.5 Hz. Fig. 11 shows the latency and loss for *where* 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 Fig. 12, 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.

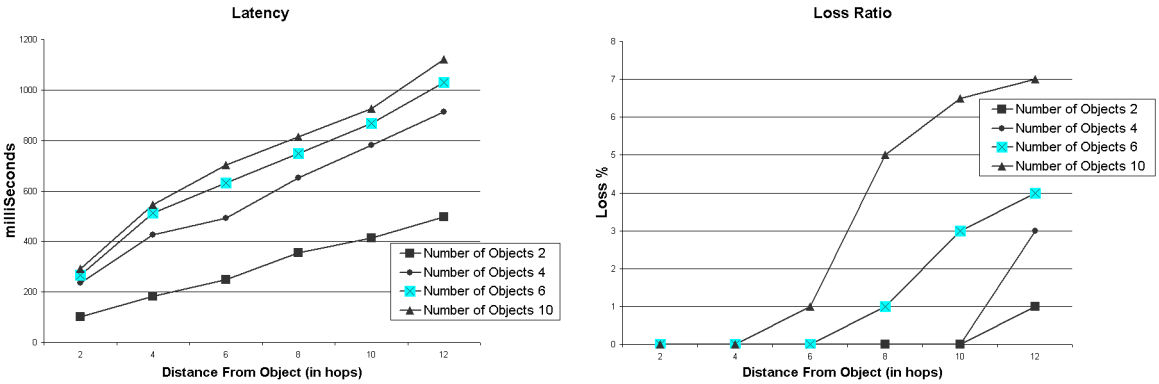
Scaling in object speed: Fig. 14 shows the latency and loss for *where* operations with increasing object speeds that generate updates at 0.33 Hz, 0.5 Hz and 1 Hz. The query frequency is 0.5 Hz and the number of objects is 6.

Evaluation: We observe from the above figures that *Trail* offers a query response time that grows linearly with the distance from an object. Scaling the number of objects up to 10 yields a loss rate of up to 7% with a query



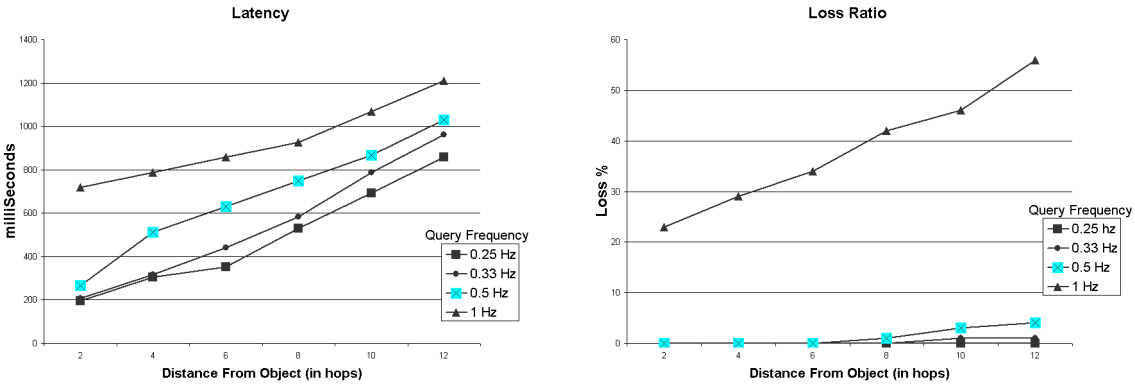
(a) Latency of *Trail* (b) Loss Ratio

Figure 10. Trail: Scaling in Number of Objects (Query frequency 0.33 Hz, Object Update 0.5 Hz)



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

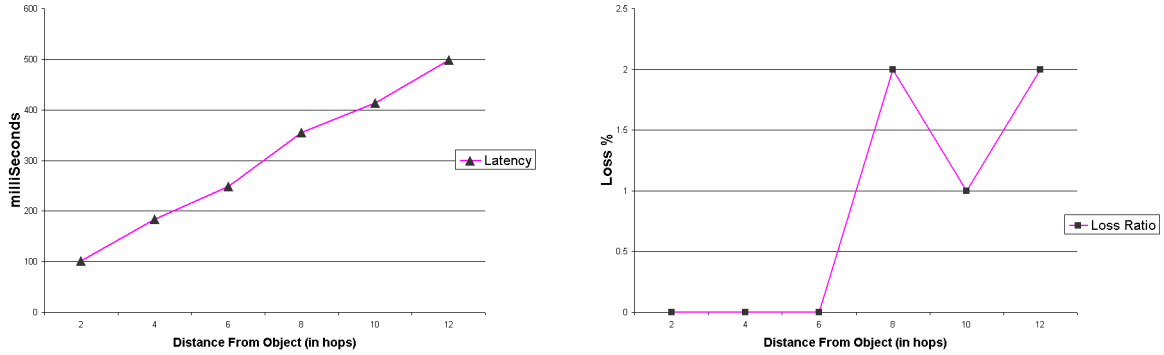
Figure 11. Trail: Scaling in Number of Objects (Query frequency 0.5 Hz, Object Update 0.5 Hz)



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

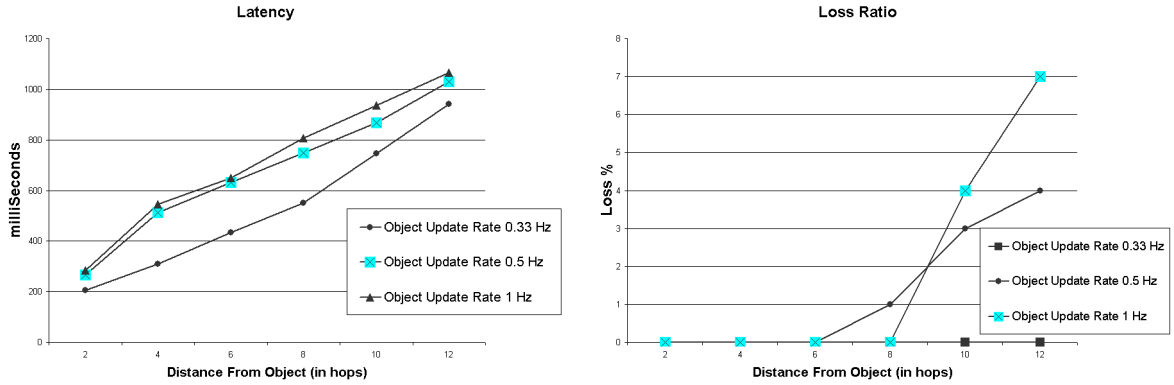
Figure 12. Trail: Scaling in Query Frequency (6 Objects, Object Update Rate 0.5 Hz)

frequency 0.5 Hz and an object update rate of 0.5 Hz. Scaling the object speeds to generating 1 update per second results in a loss rate of up to 7 % even with 6 objects in the system and query frequency of 0.5 Hz. We notice that latency and loss increase substantially as the query frequency becomes 1 Hz; this happens due to higher interference leading to congestion. Even at lower distances, there is significant loss because all the 6 objects are within interference range. As seen in Fig. 13, the loss ratio of *Trail* with 2 objects under query frequencies of 1 Hz with object update rate of 0.5 Hz is much lower. Applications may compensate for losses by increasing their query frequency, but this should account for extremal scenarios where the increased frequency itself results in higher interference. By the same token, applications should be aware of other extremal conditions (in terms of object number and speed) for effectively using the service. And, application design should attempt to increase loss tolerance.



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

Figure 13. Trail: Scaling in Query Frequency (2 Objects, Object Update Rate 0.5 Hz)



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

Figure 14. Trail: Scaling in Object Speed (6 Objects, Query frequency 0.5 Hz)

5 Trail in the Synchronous Model

Although *Trail* can find the location of mobile objects in time and work proportional to distance from the object, it follows an asynchronous model for the queries for which the radios of motes in the network have to

be always awake to support the queries. This is an energy consuming operation. In this section we describe *Synchronous Trail*, a network service in which the algorithm for *where* and object update operations are exactly like in *Trail*, but the network operates synchronously, based on the ideas drawn from *Trunk*. We thus gain energy efficiency. We evaluate how the latency of *where* operations is affected.

Description: The network of length L is divided into smaller segments of m clusters each. In alternate segments, message waves are scheduled along the backbone in both the directions. In those segments where the backbone waves are not scheduled, messages are aggregated from the non-backbone nodes and sent to the backbone nodes in a procedure similar to *Trunk*. At the end of the backbone wave time in a given segment, the aggregation is performed in the segment and the backbone wave moves to the neighboring segments. This is illustrated in Fig. 15.

We now analyze the minimum required m so that the backbone waves can proceed to the neighboring segments without incurring a delay at segment boundaries and still not interfere with the aggregation in the neighboring segment. Recall from our analysis in *Trunk* that the time required for a wave to traverse a segment of length m in both directions is l slots. This is taking into account the staggering required at the center of the segment. We require that the updates in the neighboring segment be completed before the wave enters the communication range of the neighboring segment so that the waves do not incur a wait at the boundary of the segments. Out of the l time slots, during $m-2$ time slots, there is no interference with communication in the neighboring segment. Also recall that if there are n objects in the system, the time required to send the aggregate for n objects from non-backbone nodes to the backbone is $n+2$ slots. Thus we require that $m-2 > n+2$ or $m > n+4$ in order for the waves to not incur a wait at boundary segments.

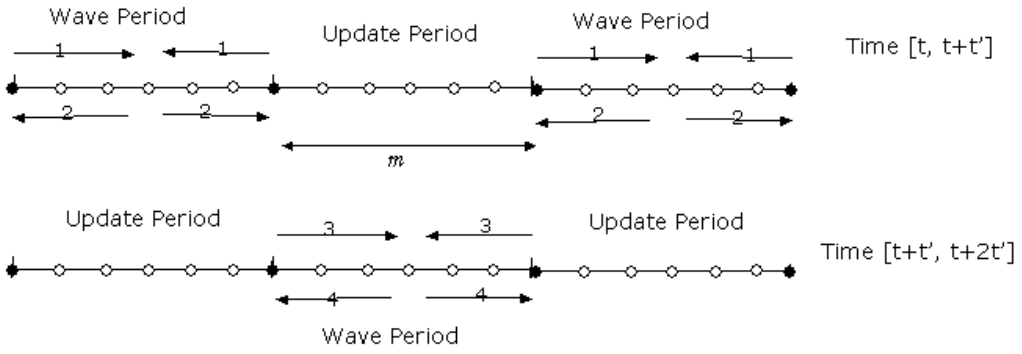


Figure 15. Synchronous Trail Operation

However, it is a conservative choice for the segment size to depend on the total number of objects in the system. Instead, if we assume an upper bound on the number of objects within a region, we can decrease the required segment size m. The minimum required segment size for the backbone waves to not incur a wait at the segment boundaries is the smallest number m such that there are at most n' objects in m clusters and $n' < m-4$.

For example, if it is known that there can be at most 2 objects within one cluster, it is not suficient to satisfy the condition stated in the above lemma. However, if it is known that there can be at most 9 objects within any region of 14 consecutive clusters, then we can satisfy the condition stated in the above lemma and m can be 14 or more.

We now analyze the energy and latency aspects of *Synchronous Trail*. The analysis for energy is similar to *Trunk*. The backbone wavetime per segment is $m\delta$ time. Each backbone mote is awake for 1 slot in the backbone wave time per segment to listen to aggregated message from non-backbone mote, 2 slots for listening to wave message from neighboring backbone and at most 2 slots to transmit to neighboring backbone mote. The central backbone mote in every segment is awake for 4 slots in every wave time to listen and at most 2 slots for transmitting.

We now analyze the latency. The messages that are being transmitted are the messages for *Trail* including *where*, *here*, *grow*, and *clear*. In *Trail*, the latency is equal to the transmission time from source to destination and is proportional to the number of hops. In *Synchronous Trail*, the proportionality is maintained but due to the synchronous operation, each message incurs three additional types of delay: *aggregation delay*, *pickup delay* and *center delay*. The *aggregation delay* is the time during which the messages are being aggregated from non-backbone to backbone motes. Note that this delay is equal to the wave period of a segment. This is a fixed delay of m time slots. The *pickup delay* for a message is the delay incurred after the aggregation until the wave from the neighboring segments reaches the backbone mote where the message has been aggregated. Depending on the position of a mote within a segment, this delay is between 1 and m. The *center delay* is the delay introduced due to staggering of the messages at the center. This delay is 1 hop for every segment that the message passes through. Hence we have,

Lemma 5.1. *In* Synchronous Trail*, the latency for an object p receiving a reply to a query* $\mathbf{where(obj_i, obj_p)}$ *is given by Eq. 1, where m is the number of clusters per segment, s is the number of segments that the* where *query passes through and dist(x,y) is the hop distance between motes x and y.*

$$Latency \quad = \quad 2 \times (dist(i.agent, p.agent) + 2m + s) \tag{1}$$

Thus, we see that there are energy and latency tradeoffs involved with the choice of the length of each segment. As the length of each segment increases, the latency increases but we also get increased energy efficiency as each backbone node can sleep for longer.

For reasons of space, we relegate the experimental evaluation of the latency and reliability of *Synchronous Trail* to an extended version of this paper.

6 Related Work

Tracking: As mentioned earlier, mobile object tracking has received significant attention [3, 6, 9, 13, 14] and we have focused our attention on WSN support for tracking. Some network tracking services such as [6] have nonlocal updates, where updates to a tracking structure may take work dependent on the network size rather than distance moved. There are also hierarchical solutions such as [3] where a hierarchy of regional directories is constructed and the communication cost of a find for an object d away is $O(d * log2N)$ and that of a move of distance d is $O(d * logD * logN + log^2 D * logN)$ (where N is the number of nodes and D is network diameter). A topology change, such as a node failure, however, necessitates a global reset of the system since the regional directories depend on a non-local clustering program that constructs sparse covers. *Stalk* is a self-stabilizing hierarchical tracking service for sensor networks that supports locating a mobile object in time and work proportional to the

distance from the object. *Trail* uses a tracking structure similar to *Stalk*, but by considering a linear backbone topology, uses only one level of hierarchy in clustering. Whereas *Stalk* is developed for only an asynchronous model, we discuss how *Trail* can be implemented in a synchronous model, so as to make it more energy efficient by having a synchronous schedule along the backbones.

Association: Our network tracking services assume the existence of an object detection and association service that detects the presence of an object, associates it with the previous detections for the same object and assigns an id. This is an orthogonal service to object tracking/assignment, and as such may be considered separately. Association services can be implemented in a centralized [21] or distributed (using handoff mechanisms) [20] fashion; the latter approach would suit integration with *Trunk* and *Trail*.

Querying and storage: Querying for events of interest in WSNs has also received significant attention [10,12, 16,19]. In directed diffusion, a tree of paths is created from all objects of interest to the tracker. 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 structure on the network and tracks to all objects are maintained on this structure rooted at a point. Thus, updates to the structure are local and any object can find the state of any other object following the same tracking structure.

The question of when to push versus pull is relevant in the context of this paper. In [11], a balanced push-pull strategy is proposed that depends on the query frequency and event frequency. 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 which is not a goal of [11]. Moreover, while the strategies in [11] are intended for a snapshot service they do not ensure a consistent snapshot, and would yield higher communication context in the use case of *Trunk*.

There has also been considerable work on data-centric storage where the focus is on efficiently placing data at precise location providing easy access. The property of distance sensitivity is again not a goal in that work. This motivates why we maintain the state of objects only in a node closest to the object. In *Trail*, we maintain pointers to the current location that are updated in cost and time proportional to the distance moved. By doing so, we get a latency for *where* operations that decreases as the object being tracked becomes closer. In *Trunk*, the state of mobile objects is pushed to all subscribers at the requested frequency.

Scheduling: The idea of scheduling node transmission/listening for energy efficiency during data gather operations has been explored before in the context of sensor database systems [22]. In contrast to that work, to reduce latency, we schedule based on the number of objects that send an update and are within an interference range, as opposed to the number of motes within interference range. Moreover, our scheduling is at the middleware level as opposed to the MAC layer

7 Conclusions and Future Work

In this paper, we presented the *Trunk* and *Trail* network services for supporting different aspects of distributed object tracking. *Trunk* is a snapshot service that returns consistent, global snapshots containing the state of all objects in the system to all subscribers in an energy efficient way. It enables applications such as *PEG* and

IIG to efficiently assign evaders/intruders to pursuers/interceptors. Its model is synchronous and periodic, and its latency increases as the network length and number of object increases. By following a schedule that avoids interference among the motes, it achieves high reliability. By imposing a TDMA schedule based on the number of objects in the system rather than the number of motes in the network, which could be much larger, it reduces the latency of gathering the snapshot. In contrast, *Trail* works in an asynchronous model and returns the location of any mobile object to an object issuing the query, in time proportional to the distance between the objects. It enables applications such as *PEG* and *IIG* to track a particular object where the interceptor applications require information about intruders faster as they come closer. By communicating only in the region between the two objects, it operates in a local and energy efficient manner. Since it works in an asynchronous model where the application can issue a query at any time, motes are always *on* and there is more interference, by exploiting ideas of *Trunk*, we are able to develop a synchronous version that improves its energy efficiency and reliability at the cost of latency.

We also presented experimental results of the performance of *Trunk* and *Trail* in a wireless sensor network testbed of 105 XSMs using mobile object traces from an outdoor experiment. We characterized the message loss in these services as a result of residual communication interference. It may be of interest to characterize in future work models of these losses that application can exploit so as to select service times and their rates of operation. Conversely, giving the network services knowledge of the state of the application, such as deadlines for query replies or estimated location of objects in the network, may yield the network services that further improve the reliability and latency. Moreover, it may also be of interest to study the sensitivity of diverse applications to loss rates to support appropriate selection of controllers; this will handle situations where the losses increase due to congestion resulting beyond a certain query frequency which may occur if many objects come close together.

Although we have presented *Trunk* and *Trail* for a linear topology of the backbone motes, there is a straightforward extension of these services if the backbone motes form a star topology spanning a two-dimension plane with a non-trivial angle between the spokes of the star. As the network scales in size, at some point it may no longer be possible to form such a star topology that covers the entire network. For such networks, again there is a ready extension of these services that is based on the hierarchical clustering of the network suggested in [14]. We are interested however in the open problem of how to implement distance sensitive object tracking in a large network without using hierarchical clustering and this is a subject of ongoing work [5].

References

[1] A. Arora, R. Ramnath, E.Ertin, S. Bapat, V. Naik, and V. Kulathumani et al. Exscal: Elements of an extreme wireless sensor network". In *The 11th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2004.

[2] Anish Arora, Prabal Dutta, Sandip Bapat, and Vinodkrishnan Kulathumani et al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks, Special Issue on Military Communications Systems and Technologies*, 46(5):605–634, July 2004.

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

[4] H. Cao, E. Ertin, V. Kulathumani, M. Sridharan, and A. Arora. Differential games in large scale sensor actuator networks. Technical report, The Ohio State University, 2005.

[5] M. Demirbas and A. Arora. A light weight querying service for sensor networks. Technical Report 2005-24, State University of New York at Buffalo, 2005.

[6] S. Dolev, D. Pradhan, and J. Welch. Modified tree structure for location management in mobile environments. In *INFOCOM*, pages 530–537, 1995.

[7] E.Ertin, A.Arora, and S.Bapat. Hybrid sensor network experiment with osu kansei testbed. In *Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[8] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[9] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Gang Zhou, Jonathan Hui, and Bruce Krogh. Vigilnet:an integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks*, 2004.

[10] C. Intanogonwiwat, R. Govindan, D. Estrn, J. Heidamann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE Transactions on Networking*, 11(1):2–16, 2003.

[11] Xin Liu, Qingfeng Huang, and Ying Zhang. Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks. In *ACM Sensys*, 2004.

[12] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, 2003.

[13] M.Demirbas, A.Arora, and M.Gouda. A pusuer evader game for sensor networks. In *Symposium on Self Stabilizing Systems (SSS)*, 2003.

[14] M.Demirbas, A.Arora, T.Nolte, and N.Lynch. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *8th International Conference on Principles of Distributed Systems (OPODIS)*, 2004.

[15] V. Mittal, M. Demirbas, and A. Arora. Loci: Local clustering service for large scale wireless sensor networks. Technical Report OSU-CISRC-2/03-TR07, The Ohio State University, 2003.

[16] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S.Shenker. Ght: A geographic hash table for data-centric storage. In *Wireless Sensor Networks and Applications (WSNA)*, 2002.

[17] S.Bapat, V. Kulathumani, and A.Arora. Analyzing the yield of exscal, a large scale wireless sensor network experiment. In *13th IEEE International Conference on Network Protocols*, 2005.

[18] S.Bapat, V. Kulathumani, and A.Arora. Reliable estimation of influence fields for classification and tracking in an unreliable sensor network. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2005.

[19] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. In *First ACM SIGCOMM Workshop on Hot Topics in Networks*, 2002.

[20] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target indentities in wireless ad hoc networks. In *Second International Conference on Information Processing in Sensor Networks*, 2003.

[21] B. Sinopoli, C. Sharp, L. Schenato, and S. Sastry. Distributed control applications within sensor networks. In *Proceedings of the IEEE*, volume 91, pages 1235–46, Aug 2003.

[22] A. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Wavescheduling: Energy efficient data dissemination for sensor networks. In *International Workshop for Data Management in Sensor Networks*, 2004.