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[3], and Anish Arora[1]

[1]Dept. of Computer Science and Engineering, The Ohio State University
[2]Dept. of Computer Science, State University of NewYork at Buffalo
[3]Dept. of Electrical and Computer Engineering, The Ohio State University

Abstract

In this paper we design two wireless sensor network services Trunk *and* Trail*, for distributed, mobile object tracking applications. Both these services are designed to be reliable and energy efficient while satisfying their specification. The specification for Trunk is to return a consistent global snapshot of the state of all objects in the system to all subscribed applications at regular intervals of time. Because the queries are specified apriori and are periodic* Trunk *operates in a synchronous model, and therefore the nodes in the network listen on radio and transmit only when scheduled, thus being energy efficient. The specification for* Trail *is to return the location of a particular mobile object to any application issuing the query.* Trail *maintains a tracking data structure for all objects and this structure is updated locally upon object moves and using this structure* Trail *offers a find time that increases linearly with the distance from an object. However* Trail *operates in an asynchronous model where the different application queries are not synchronized and nodes in the network are always awake to listen queries. Since this is an energy consuming, we design a* Synchronous Trail *service in which the algorithms for maintaining the tracking data structure and the* find *operations are like in* Trail*, but the nodes in the network operate synchronously and thereby gaining energy efficiency at the cost of latency. We then consider an example mobile object tracking system, called an intruder-interceptor game and describe how the interceptor application uses our network services* Trunk *and* Trail*. We also provide an experimental analysis of the performance of these two services in terms of their reliability and latency in a network of 105 XSMs (eXtreme Scale Motes) using* Kansei*, a wireless sensor network testbed.*

1 Introduction

Tracking of mobile objects has recently received significant attention in the context of military applications, mobile computing and cellular telephony. Mobile object tracking can be done either by forming an ad-hoc network of the mobile objects themselves or a network of static devices could provide the sensing and communication infrastructure for tracking of the mobile objects. In this paper we focus on mobile object tracking using a wireless network of static sensors. Some examples of object tracking systems in the context of military requirements using sensor networks are Line in the sand [2, 12] and Exscal [1, 11]. These and other tracking systems using sensor networks thus far have focused mainly on applications that monitor the tracks of objects. In this paper we design services that allow *closing the* *loop* and enable control based tracking applications. Examples of control based tracking applications are pursuer-evader tracking and intruder-interceptor tracking.

Control based tracking applications could be centralized or distributed in nature. By tracking *application*, we mean the component of the system which based on the tracks of the various objects in the system, computes the control strategy. There have been demonstrations of pursuer evader tracking systems using sensor networks. However, the architecture for these systems have been such that the tracking applications are centralized or semi-centralized in nature. However, object tracking systems especially with control based applications face extreme scale issues in terms of the size of the network and speed of objects being tracked. Hence it is desired that these applications are distributed in nature.

Different application scenarios arise in multi object tracking systems. For example, in a pursuer evader tracking system, pursuers are interested in tracking evaders such that some optimization criteria is met such as minimizing the average catch of time of all evaders or maximizing the distance of evaders from a valuable asset. The latter application is also called an intruder-interceptor game. One common application scenario that arises in these system is to make an optimal assignment of intruders to interceptors. This applications uses global knowledge of the system to make the optimal assignments and therefore these applications require consistent snapshot of the system at regular intervals. These queries by way of being periodic and specified upfront suggest a push model for the underlying network tracking service. Another application scenario in this system arises when an object is being tracked by an interceptor. Nash equilibrium conditions for satisfying the optimality constraints imply that in such cases, the rate at which the application requires information about the trackee is not constant. The frequency at which information is needed depends on the relative location of the objects. This suggests a pull model for the underlying network tracking service.

While sensor networks provide opportunities for tracking of objects, they also use energy-constrained devices. Communication is a significant source of energy drain in these devices. Transmitting on the radio or simply listening on the radio decreases the battery life of these devices. These requirements motivate the need for network services that support the applications to be energy efficient. Algorithms that require extensive communication are not desirable. Sensor networks are also prone to node failures and message losses and thereby can lead the programs to arbitrary states. The objective of this paper is to design network services for object tracking systems, that are reliable and energy efficient.

Contributions: In this paper we first describe an architecture for distributed, mobile object tracking systems using sensor networks. We then design two network services *Trunk* and *Trail* for tracking mobile objects. Both these services are designed to be energy efficient while satisfying their specification. The specification for Trunk is to return consistent global snapshot of the state of all objects in the system to all subscribers at regular intervals. Since the queries are specified upfront and they are periodic, Trunk follows a push model and also operates synchronously. By being synchronous, nodes in the network listen on the radio or transmit only when scheduled thus giving energy efficiency. The specification for Trail is to return the location of a particular object to the client issuing the query. Trail maintains a tracking data structure by propagating mobile object information. Trail also offers a distance sensitive property. The time taken to complete the find operations is proportional to the distance between the objects. Also the time and work to update the tracking structure is proportional to the distance moved. By operating in a local manner, Trail is energy efficient. However, Trail is an asynchronous protocol 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 *find* operations are exactly like Trail, but the nodes in the network operate synchronously. In all these services, the client application itself could be a mobile object in which case the replies are returned to the object at its current location.

We then consider an example distributed multi object tracking application called an intruder-interceptor game. The application works in two phases, assignment and tracking. During assignment, the applications use global knowledge of the system and during tracking applications use local knowledge to intercept the intruders. We show how our network services *Trunk* and *Trail* can be used to satisfy the requirements imposed by these phases respectively in order to succesfully intercept the intruders. We also provide an experimental analysis of the performance of these two services in a network of 105 XSMs using *Kansei*, a wireless sensor network testbed [6].

Organization of the paper: The rest of the paper is organized as follows. In Section 2, we describe the system architecture. In Section 3, we describe the network and fault model and formally state the specification of the network services, *Trunk* and *Trail*. In Sections 4 and 5, we describe our services *Trunk* and *Trail* respectively and present the results of experimental evaluation of their performance in a wireless sensor network testbed. In Section 6, we describe *Synchronous Trail* and in Section 7, we describe how *Trunk* and *Trail* can be extended to 2-dimensional network topologies. We conclude in Section 8.

2 System Architecture

In this section we describe an architecture for distributed mobile object tracking systems. The system comprises mobile objects, and a network of static sensors. Tracking applications (which could be running on a subset of the mobile objects themselves) use the sensor network to track desired objects. The Fig. 1 shows the overall architecture of the system and how the components are interconnected.

Each node in the network consists of a sensing component and a radio component. Each node also participates in three distributed network services namely, object detection and association, network tracking and time synchronization.

The object detection and association service uses the sensing and radio components to locate an

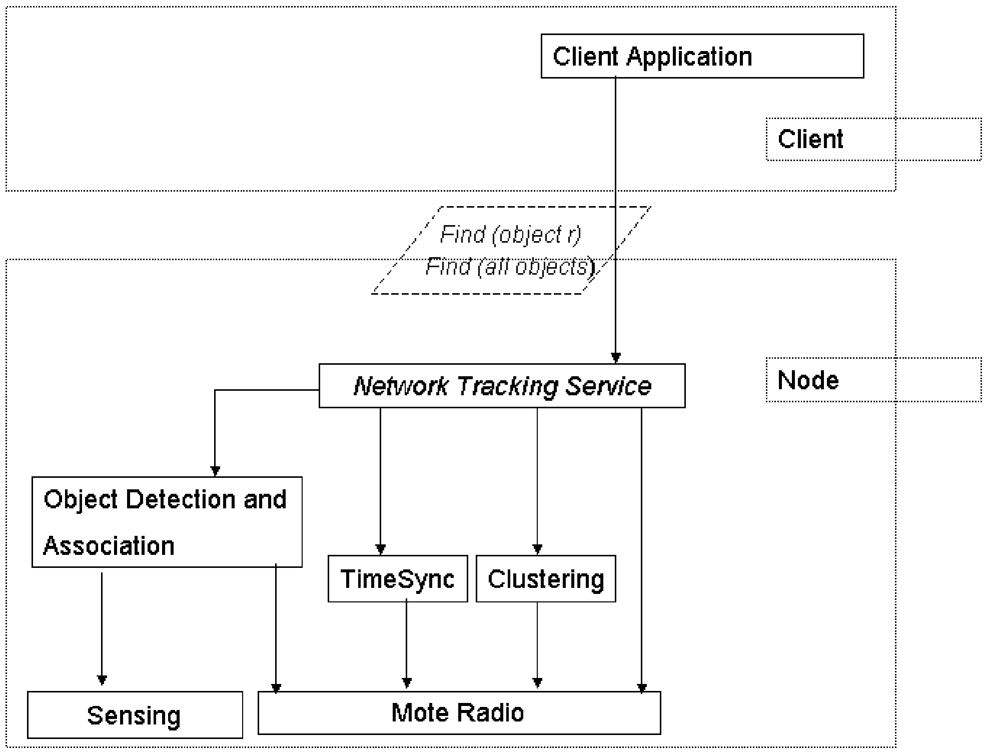


Figure 1. Fig. 1: Architecture of Mobile Object Tracking System

object and also corroborate object detections with previous detections for the same object. This service also assign a unique id in the range $1..n$, where n is the number of objects in the system, when they are first detected. We now define an *Object agent*.

Definition .1 (Object Agent) *Among all the nodes that sense a mobile object, the agent for an object agent at any instant is the node that is closest to the object at that instant.*

The object agent stores the state of the object which includes variables such as the location of the object. Note that the location of an object could be in a different space and more fine grained than the location of the sensor nodes themselves.When an object moves, the state of the object is transferred to the new agent. In this paper, we assume the existence of such an object detection and association service.

The network tracking service is responsible for tracking the mobile objects across the network. Informally, the specification for this service could be where is object i or find the nearest object or find all objects in the system. In this paper we focus on network tracking services for multiple mobile objects. We design two such services Trunk and Trail that have different specifications. The objective of both these services is to minimize the energy used while satisfying their specifications.

The time synchronization service across the network may be needed for certain tracking services.

For example, Trunk uses a time synchronization service but Trail does not. Time synchronization could be implemented in a number of ways. The nodes could be GPS enabled or the nodes could run a periodic distributed beaconing protocol. Time synchronization could also be implemented using an explicit initialization process and assuming that clocks at all processes move at the same rate and new nodes that add into the network overhear messages to synchronize with the network. For simplicity, Trunk assumes that a time synchronization service exists in the network.

Both Trunk and Trail use a network clustering service that forms a backbone for the network such that all nodes in the network are within communication range of their nearest backbone node. Moreover we assume that nodes within a cluster are in radio range of the neighboring cluster but not beyond that.

An example of such an object tracking system is that of intruder-interceptor tracking. The mobile objects are interceptors and intruders. The clients to the network services are the tracking applications running on the interceptor objects. In this paper, we consider two example interceptor applications, GlobalOne and LocalOne and show how our tracking services Trunk and Trail are used by these applications respectively. GlobalOne is used to make optimal assignment of interceptors to intruders and all the interceptors decide their next step based on this global information. In LocalOne, the interceptors track a particular evader and they require information about a subset of objects.

The object detection and association service enables the formation of pursuer and evader agents. The pursuer agent acts as the communication interface for the pursuer with the network. In the following section, we describe the system model and formally describe the specification for the above services.

3 Model and Specification

In this section, we describe the system setting and fault model and formally state the specifications for each of the network services described in the previous section including the tracking services, Trunk and Trail.

3.1 System Setting

In this subsection, we describe the network model, state the notations that we use in the paper and provide an abstraction for the underlying message passing.

Network Model: We consider a sensor network with multiple sensor locations. The sensor locations are partitioned into L clusters with a cluster-head for each cluster. These cluster-heads form the backbone of the network. We assume in this paper that the backbone nodes are arranged in a linear topology. All nodes within a cluster are assumed to be within communication range of each other and their respective clusterhead. There are upto n mobile objects in the network.

Notations: As a convention, $j.r$ refers to a variable r at sensor node j. $j.bbid$ refers to the id of the

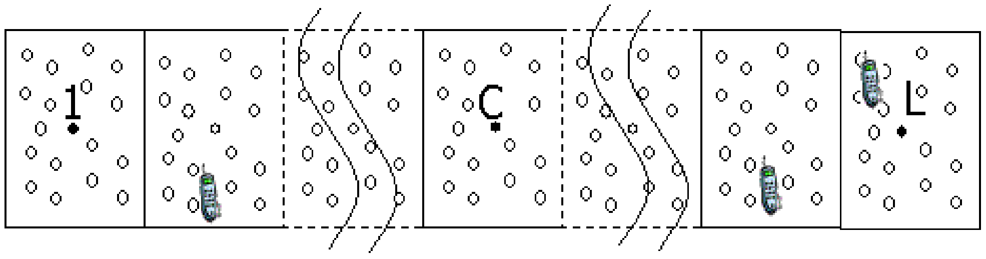


Figure 2. Network of Sensor Nodes and Mobile Objects

cluster in which node j belongs to. We denote one cluster in the network as a central cluster and its cluster-head is denoted as C. Note that C need not be physically at the center of the network. Every node 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 nodes and backbones farthest from the center, $j.dnbr$ is set to $\perp$. Also $C.unbr = \perp$. Note that $dist(x, y)$ returns the hop distance between x and y. Two non-backbone nodes within a cluster have hop distance 2 and two node in clusters r apart have hop distance $2 + r$. Variables specific to the network services, Trunk and Trail are defined in the respective sections. Objects are denoted by their ids obj_p. Location of an object p is given by $< obj_p.x, obj_p.y >$. Also, $obj_p.r$ refers to a variable r at object obj_p.

Communication Abstraction: Communication between nodes i and j are achieved by using $\mathbf{send(m)_{i,j}}$ and $\mathbf{receive(m)_{i,j}}$. Nodes i and j could be multi-hop. $\mathbf{send(m)_i}$ is used by node i to send a message within to any node within 1-hop of i. Similarly $\mathbf{receive(m)_j}$ can be used to receive messages sent within 1 hop of j. δ is assumed to be the 1-hop message transmission time.

3.2 Network Services Specification

We now state the formal specifications for each of the network services in the previous section. In this paper we focus on the design of network services for tracking.

Object Detection and Association Service: Each mobile object is detected by all sensor nodes within the sensing range. The object detection service returns an event $\mathbf{detected_p}$ at the node that is closest to the object p. The signal contains the state of all variables associated with object p. This node is called as the agent for object p. The service also returns an event $\mathbf{moved_p}$ at the previous agent for object p unless p has been detected for the first time in the network. The object detection service also assigns a unique id p in the range $1..n$ to every object in the network.

Time Synchronization Service: This service ensures that all the sensor nodes are synchronized to a global time. A node can get the current global time using **GetGlobalTime**().

Trunk Service: The trunk service returns a consistent global snapshot of the state of all mobile objects. Trunk answers client queries of the form $\mathbf{get(GlobalSnapshot, T)}$ where T is the period at which the snapshots are needed. Ecah client gets an identical snapshot of the system.

Trail Service: Trail offers the following function: $\mathbf{where(objecti, objectp)}$. $\mathbf{where(objecti, objectp)}$ 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 sensitive property. The time taken to complete the find operations and the amount of work for these operations is proportional to the distance between the objects. Also the time and work to update the tracking structure is proportional to the distance moved.

The objective of both these services is to be energy efficient while satisfying their specifications. We now describe the two services and in each case describe how pursuer evader tracking applications use the service.

3.3 Fault Model:

The network can corrupt or lose a message if it interferes with any other message sent at the same time. Therefore nodes can suffer arbitrary state corruption. A system is said to be self stabilizing iff starting from an arbitrary state the system eventually recovers to a consistent state, a state from where its specification is satisfied. We characterize consistent states for our services in the following sections.

4 Related Work

Tracking of mobile objects has recently received significant attention in the research community. In this paper we focus on mobile object tracking where the sensing and communication infrastructure is provided by a network of static sensors and we design network services for tracking.

There exist network tracking services such as [5] that suffer from nonlocal update problems, 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] for mobile object tracking, in which 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). But a topology change, such as a node failure, 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 find of a mobile object in time and work proportional to the distance from the object. *Trail* uses a tracking structure similar to *Stalk*. In *Trail* we have only one level of hierarchy in clustering and we maintain tracks for multiple objects with all tracks are rooted at one point. In this paper, we also discuss how *Trail* can be made more energy efficient by having a synchronous schedule along the backbones. The applications that request the *find* operation themselves could be mobile objects being tracked in which case the results are returned to the current location of the objects.

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 problem to object tracking and could be done in a centralized manner [15] or in a distributed manner using handoff mechanisms [14].

Querying and Storage in Sensor Networks: There has been significant research on querying for event of interest in sensor network [7, 9, 10, 13]. In directed diffusion, a tree of paths is created from all objects of interest to the tracker. These paths have to be updated when any of the objects move. A change in assignment would require creation of new paths. On the other hand, in *Trail* we impose a fixed structure on the network and tracks to all objects are maintained on this structure rooted at a point. Updates to the structure are local and any object can find the state of any other object following the same tracking structure.

There has also been considerable work on data centric storage where the focus is on efficiently placing data at precise location providing easy access. In *Trunk* and *Trail*, we maintain the state of an object ony at the node closest to the object. In *Trail*, we maintain pointers to the current location that is updated in cost and time proportional to the distance moved. By doing this, we get a latency for *find* operations that decreases as the object being tracked is closer. In *Trunk*, the state of mobile objects is pushed to all subscribers at the requested frequency.

This leads to the discussion of push vs pull model [8] for object tracking applications. In case of *trunk*, since the queries are periodic in nature, we follow a push model and also make it energy efficient by being synchronous. In case of *Trail* since the query frequency is varying, we adopt a pull model, but by maintaining the tracking structure we ensure that finds are not global operations. We also discuss how to implement this asynchronous query model using an underlying synchronous schedule for the nodes.

The idea of scheduling nodes to transmit and listen in order to get energy effciency during data gather operations has been explored before in the context of sensor database systems [16]. However, in this paper we impose the schedules at the middleware level and not at the MAC layer. Moreover, by scheduling based on the number of objects that send an update and are within an interference range rather than the number of nodes within interference range, we are able to decrease the latency introduced by scheduling.

5 Trunk

Trunk is a network service that executes a synchronous protocol for disseminating a global snapshot of the system to all subscribers. The specification for 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.

Since the snapshot is required at a prespecified constant rate, the implementation for Trunk follows a *push* model. *Trunk* exploits the knowledge of the period T and schedules the collection of individual object snapshots and distribution of the global snapshot. By being synchronous, sensor nodes can *sleep* unless they sense an object or it is their turn to listen or transmit object snapshots and thus is energy efficient.

5.1 Description

Trunk assumes the existence of a time synchronization service and the notion of a global time. Recall that the specification for the time synchronization service is of the form **GetGlobalTime**(). Each node j stores the global time as *j.t*, that is continuously updated. For simplicity of explanation, let us assume that the network has $L = 2l + 1$ clusters with l clusters on either side of the central clusterhead C. This is shown in the following figure. Let *j.pos* denote the relative position of node j's cluster with respect to C. The value of *pos* ranges from -1 to $-l$ one one side of C and 1 to l on the other side. For nodes on the central cluster $j.pos = 0$. Let *j.unbr* denote the neighboring backbone node towards the center and *j.dnbr* denote the neighboring backbone node away from the center. For non-backbone nodes, *j.upnbr* in the nearest backbone node, the cluster head. If and only if a node j is an agent for object i, a $\mathbf{detected_i}$ event is raised, and $j.detected_i$ is atomically 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 $j.snapshot_i(ts)$. An object i's view of the global snapshot is given by *k.globalsnapshot*. This is again the union of the state of all objects in the system. Every subscribed object gets the global snapshot once at interval boundaries sT where s is an integer, from its agent node at that instant.

The period T for the snapshot should be atleast long enough to accomodate an *update period* and a *wave period*. During the update period, individual object updates are sent from the non backbone nodes to the nearest backbone node. During the wave period, the backbone nodes gather their individual snapshots and disperse to the subscribers.

We first describe the actions during an update period. When global time equals the start of the update period, for all objects i for which $j.detected_i$ is set, node j records the state of i in $j.snapshot_i$. During the update period, the non-backbone nodes within a cluster gather the snapshots within the cluster and send to the backbone nodes. Since *Trunk* has knowledge that there are atmost n objects in the system, n slots are reserved for the updates to be gathered among the non-backbone nodes and sent to the backbone node. A non backbone node sets *j.maxi* as the highest id of the object for which $j.snapshot_i$ is not null and sets itself as the leader. A node with $j.maxi = p$, sends *j.snapshot* to all nodes in the cluster in the p^{th} slot. Nodes in the cluster with *j.maxi* higher than p append this to their snapshot and send it during their slot. Node with update for object with highest id in the cluster is responsible for sending the snapshot to the nearest backbone node. We call this node the *leader* for the cluster for that interval. Thus we ensure that in every period T, there is atmost one update message to a backbone node. Note that the update period could be as small as a function of n' where n' is the number of objects within a cluster at any time. This can be done with communication from the application or with motion models for the objects. In *Trunk* we assume that all objects could be in the same cluster at any time.

The gathered snapshots are sent to the backbone nodes in two slots. Since nodes within a cluster can be heard at most one cluster away, there is no message interference during this process. Each backbone node is thus awake for just one slot during the update period. The duration of each slot equals the per-hop transmission period δ.

The wave period consists of two phases, a snapshot gathering phase and a snapshot dispersing phase. Each backbone node has one slot to transmit in each of these phases. During the gathering phase, snapshots from individual backbones are gathered starting from the nodes farthest from the center and going towards the center. In any slot during this phase, two backbone nodes are scheduled except the nodes one hop 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 this phase as well. Each backbone node is awake for atleast 2 slots and at most 4 slots during the wave period.

All nodes that are agents for subscribed objects, listen to the dispersion message and communicate it to the objects. Note that backbone nodes send a message during the snapshot gathering phase only if they hear an update during that period from a non-backbone node or if they receive a snapshot from the previous backbone node. Thus no messages are transmitted if there are no objects in the system.

Schedules within a period T are shown in the following figure. We now make the following definitions.

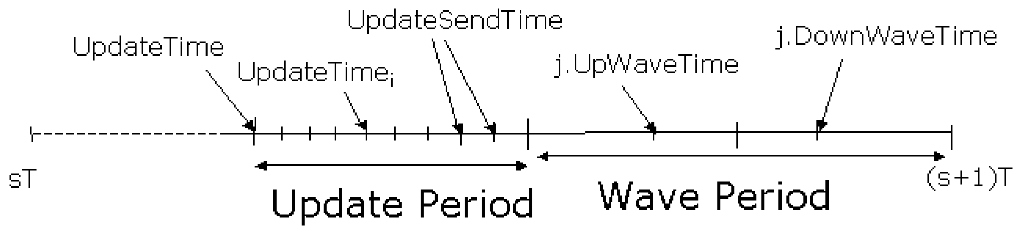


Figure 3. Schedule for nodes within an interval

Definition .2 (UpdateTime) *UpdateTime 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. UpdateTime is same across all nodes.*

Definition .4 (UpdateSendTime) *$j.UpdateSendTime$ is the time during an interval in which the leader for the cluster can send the aggregated snapshot to its clusterhead. Nodes in alternate clusters have the same $UpdateSendTime$.*

Definition .5 (UpWaveTime) *$j.UpWaveTime$ is the time during an interval in which a backbone node j can send its local snapshot to its* Up neighbor

Definition .6 (DownWaveTime) *$j.DownWaveTime$ is the time during an interval in which a backbone node j can send the global snapshot to its* Down neighbor

Definition .7 (LastUpdateTime) LastUpdateTime(t) *at any time t, is the most recent UpdateTime. It is calculated as follows: $(t/T) \times T + UpdateTime$.*

Based on the values of T, n, l, δ and *j.pos*, each node j calculates *j.UpdateTime*, $j.UpdateTime_i$, $j.UpdateSendTime$, *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 Appendix 1.

5.2 Correctness

All nodes satisfy their local invariant I at all times t. I comprises the following conditions:

- I0: $j.detect_i$ is set iff node j is agent for object i.
- I1: If $(j.snapshot_i \neq \bot)$ then
  $((j.snapshot_i(ts) = LastUpdateTime(t)) \wedge (j.snapshot_i = state_i(LastUpdateTime(t))))$
- I2: If $(\neg j.bb \wedge (((j.t\%T) < j.UpdateTime) \vee ((j.t\%T) > j.UpdateSendTime)))$
  then $j.snapshot_i = \bot$
- I2bb: If $(j.bb \bigwedge(((j.t\%T) < j.UpdateTime) \vee ((j.t\%T) > j.UpWaveTime)))$
  then $j.snapshot_i = \bot$

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 5.1 *At any time t such that $t\%T = UpdateTime$, for every object i there exists only one node j such that $j.snapshot_i = state_i(t)$.*

Proof: Only node is an agent for an object at any time and the node which is an agent at UpdateTime records the state of that object.

Lemma 5.2 *At any time t, for all objects i and all nodes j, ($j.snapshot_i = \perp$) Or ($j.snapshot_i(ts) = LastUpdateTime(t)$)*

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

Lemma 5.3 *At any time, if there exist nodes 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 $j.snapshot_i = k.snapshot_i$.*

Proof: Only one node records the state for any object in a given interval. This is the state that is passed through actions NBB_1, NBB_2 and BB_1 among the different nodes. It is also seen from the invariants that before the state of an object i is recorded in any interval in any node j, it must be that $j.snapshot_i = \perp$.

From the above lemmas, local invariants and actions of the backbone nodes, we state the following theorem.

Theorem 5.1 *For any two objects k and p, at times sT where s is an integer, $k.globalsnapshot$ is equal to $p.globalsnapshot$ and for all i in the range $1..n$, $k.snapshot_i$ equals $state_i(LastUpdateTime)$*

Proof: Since the global snapshot is sent from one single node, all nodes get the same global snapshot. In a global snapshot, the snapshots for individual objects correspond to the same timestamp and each snapshot is recorded by only node. Hence all subscribers get the same snapshot and the snapshot corresponds to the state of the system at LastUpdateTime.

5.3 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 5.2 *The state of every object i in the global snapshot $k.snapshot$ received by object k in every period T is stale by $(L + n + 3)\delta$ time.*

In each slot in the wave period, two backbone nodes transmit except one hop away from the center. the wave period 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 *k.snapshot* received by object k 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. Studies show that listening on the radio is an energy consuming operation. Trunk exploits synchronicity to be energy efficient.

Theorem 5.3 *Backbone nodes listen on radio for* $(2\delta/T)\%$ *of and transmit at most* 2 *messages in every period* T.

Theorem 5.4 *All non backbone nodes* j *that have* $j.detect_i$ *set when* $j.t = UpdateTime$*, listen on radio for atmost* $(n\delta/T)\%$ *of time.*

5.4 Fault Tolerance

We now show that Trunk is self stabilizing to its invariant conditions starting from an arbitrary state. $I0$ can be unsured by periodically checking if object i exists at j and by resetting $j.detect_i$ if not. For stabilizing to $I2$ from an arbitrary state for $j.snapshot_i$, a stabilizing action can be added that sets $j.snapshot_i$ to zero befor UpdateTime in every interval. Once this action is implemented, then using $S1$ and the above action, $I1$ is satisfied.

For message reliability in the network, *Trunk* schedules the transmissions in such a way that there is no interference. Additional reliability for the messages can be implemented in one of the folowing ways. An explicit acknowledgement scheme could be used with retransmission. Another scheme would be to use an alternate backbone 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. Both these scheme result in increasing the width of a slot to 2δ. The latter scheme also handles permanent failure of a backbone node.

Nodes 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.

5.5 Experimental Evaluation

In this section, we describe how we evaluate the performance of Trunk using *kansei*, a wireless sensor network testbed.

Experimental Setup We use a network of 105 XSM-Stargate pairs in a $15x7$ grid topology with $3ft$ spacing in the Kansei testbed. The XSMs are attached to the stargate via serial port. The XSMs also have a Chipcon radio. XSMs have a communication range of up to 40 ft and can interfere up to 60 ft under full power. The communication range can be decreased to up to 9 ft but the interference range could be higher. The statgates have a 802.11 wireless card and they are also connected via ethernet in a star toplogy to a central PC. For convinience, let us number the rows 1..7 and columns 1..15 in the testbed.

Implementation Details We let nodes 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 $3x7$ 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 6 nodes per cluster.

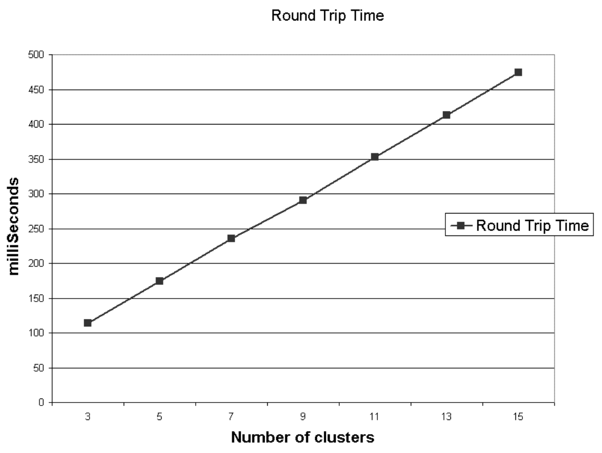
In order to minimize interference, the nodes use varying power levels as described below. During the update gathering phase, the non-backbone nodes use a power level high enough so that all nodes within a cluster can hear each other. In this phase, since the nodes transmit based on the ids of the objects, 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 feet away in $3x7$ cluater setting and 3 feet in the 6 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 approach the center.

We supply the object detection data to the network using an injector framework in the testbed. Using this framwork, we emulate an underlying object detection and association layer in order to evaluate our network services. We first describe how the object traces are obtained. 105 nodes were deployed in a $15x7$ grid topology with $10m$ 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 [**?**]. 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 $15x7$ 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. This message corresponds to the $detected_i$ event for any object i. Similarly, a *clear* message is injected at the XSM corresponding to the previous grid position, emulating the $moved_i$ message. *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 node 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 mS based on packet transmission experiments. Using the locations injected, a node also determines its *in-neighbor* and *out-neighbor*.

Performance: We now describe the experimental evaluation of 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 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. **??**, 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. **??**. 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 in the following way. Consider an object farthest from the center. Either the snapshot for a particular object or set of objects could be lost or the entire global snapshot could be lost in the network and not received at the farthest cluster from the center. Both these loss ratios are shown in Fig. **??**, as the length of the network increases. The loss ratios are over 500 snapshot intervals and 6 objects in the network. These losses could occur due to interference in the following ways. Either the updates sent from non-backbone nodes to the backbone nodes interfere because 2 hop seperated backbone nodes are within interference range or the messages interfere with low frequency time synchronization beacon messages. The former is reduced by increasing the number of slots for sending updates from non-backbone to backbone nodes to 3. The reduced loss rates are shown in Fig. **??**.



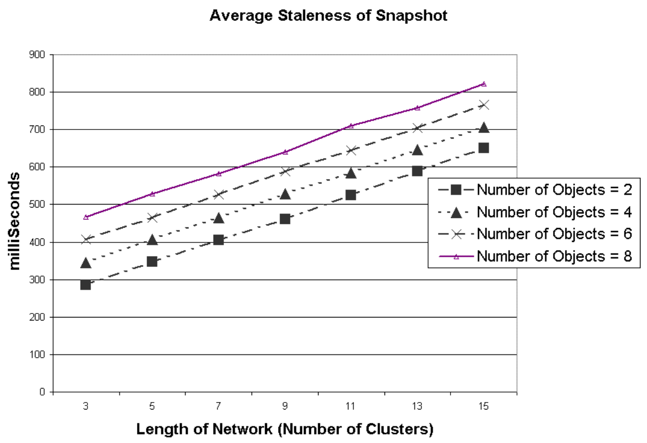
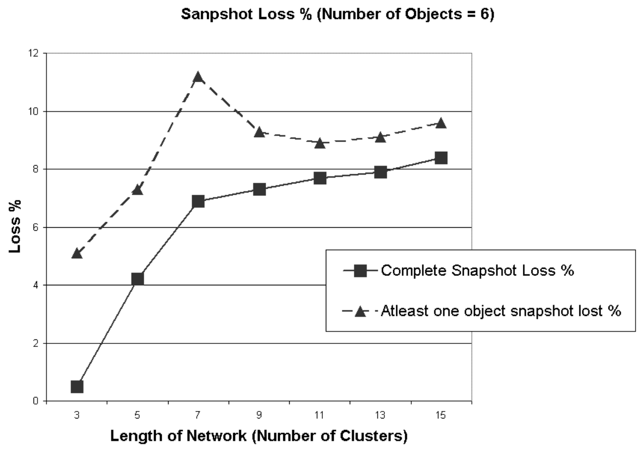
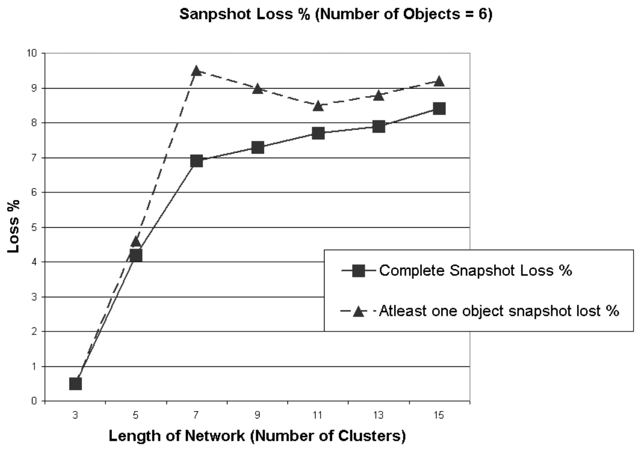


Figure 4. Average Round Trip Time and Staleness in Trunk



(a) 2 Update Slots to backbone



(b) 3 Update Slots to Backbone

Figure 5. % of Snapshots Lost in Trunk

5.6 Using Trunk for Pursuer Evader Tracking Applications

In a multiple pursuer multiple evader tracking system, often pursuer applications require a consistent global snapshot of all objects, pursuers and evaders, in order to make an optimal assignment of pursuers and evaders. *Trunk* can be used to deliver consistent snapshot at regular intervals with bounded staleness in the states of all objects. Nodes in the network that are agents for a pursuer listen to the global snapshot message at the appropriate slot depending upon their location, and send the message to the objects.

6 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: $\mathbf{where}(\mathbf{objecti}, \mathbf{objectp})$, that returns the location of object i at the current location of the object p, issuing the query. By local we mean that communication region is bounded by the location of the application isuing the query and the object being tracked. *Trail* answers the *where* queries in time proportional to the distance between the objects. To implement the function, *Trail* maintains a tracking data structure by propogating mobile object information obtained through $\mathbf{detected_p}$ and $\mathbf{moved_p}$. We first describe how the tracking data structure is maintained when the objects move.

6.1 Tracking Data Structure:

There exists a *trail* or a path for every object in the system from its current location to the center. Each process j in the system maintains pointers c_i and p_i for every object i in the system. Initially

$j.p_i$ and $j.c_i$ equals $\perp$ for all i and j. The trails are established and maintained by means of **grow** and **clear** messages.

When a **detected$_\mathbf{i}$** event is raised at any process j, if a path to object i exists at j, then a clear message is sent along that path and j sets c_i to point to itself. If a path to object i does not exist at j, then j sets p_i to point to itself and sends a **grow** message towards the center and sets $j.p_i$ to its parent along the path. When a **moved$_\mathbf{i}$** event is raised at any process j, j simply sets c_i and p_i to $\perp$.

The **grow** message propogates setting c_i towards the object until it either reaches the center or reaches a process k at which $k.c_i$ is not . When it reaches a process k at which $k.c_i$ is not $\perp$, a **clear** message is sent to $k.c_i$ and $k.c_i$ is set to the source of the grow message.

The **clear** message removes the old path to the object by following c_i and setting p_i and c_i to $\perp$ along all processes in the path. We assume that the mobile object does not relocate until an update is completed. The algorithm for maintaining the tracking data structure is shown in guarded command notation in appendix 2.

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

- I1: Iff j is agent for object i, $j.c_i = i$
- I2: If $j.c_i! = NULL$, then $j.p_i! = NULL$ or j will send a grow towards center and set $j.p_i$
- I3: If $j.p_i! = NULL$, then $(j.p_i).c_i = j$ or $j.p_i$ has sent a clear message to j
- I4: If $j.c_i = NULL$, then $j.p_i = NULL$

A tracking path for an object i is a sequence of nodes $(j_1, .., j_x, ..., C)$, 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 node that is closer to object i. A consistent state for the system with respect to an object i is one in which a tracking path exists and $j.p_i = NULL$ for every process j not in the sequence.

Following the program actions and invariant condition I1, I2 and I4, we can derive the following lemmas.

Lemma 6.1 *Starting from an initial state, if* **detected$_\mathbf{i}$** *occurs at any node j, then the system reaches a consistent state.*

Lemma 6.2 *If* **moved$_\mathbf{i}$** *occurs at any node j then $j.c_i = NULL$ and eventually there exists no process k such that $j = k.c_i$.*

Proof: Without loss of generality, let p be the node where $detected_i$ occurs, when $moved_i$ occurs at node j. Node 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 $j = k.c_i$.

Lemma 6.3 *Starting from a consistent state, 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.*

Lemma 6.4 *Starting from a consistent state, when an object i moves distance d, the number of messages exchanged to update the tracking structure to a consistent state is $dist(p, j$, where p and j are the new and old agents for object i respectively.*

Stabilization: Trail is an asynchronous protocol and the nodes are not scheduled to transmit such that collisons are avoided. Update operations for multiple objects could happen in parallel. Therefore the network can drop messages and lead the system to an arbitrary state despite reliability imposed at the communication layer. Arbitrary states are also possible when some processes restart. We now state stabilization actions for re-establishing the invariant conditions starting from an arbitrary state.

I1 is established trivially by $\mathbf{detected_i}$ event. Conditions I2 and I4 can be re-stablished by local correction. For condition I3 we use periodic **heartbeat** messages. Every node that has a valid p_i sends out a heartbeat message. If a node j has a valid p_i, but $j.p_i$ does not point to j, then this situation is corrected. Thus broken paths are re-established. The periodic **heartbeat** messages also serve to remove dead paths as shown in the following figure. If a process does not hear a grow message although $j.c_i$ is valid, j sends a **delete** message towards the center. The stabilizing actions are $s1$, $s2$ and $s3$ as shown in the figure.

6.2 Locating an object

We now describe how Trail responds to queries of the form $\mathbf{where(object_i, p)}$, where p is the client object that initiated the query. When a node receives a **where** message, if $j.c_i! = NULL$, j sends the message to $j.c_i$ or else the message is sent towards the center. We now state the following lemmas:

Lemma 6.5 *If the object i exists in the network, the trail of object i is met in at most dist $(k.agent, C)$ hops.*

The object i is said to be located when the $\mathbf{where(objecti, p)}$ message reaches process r such that $r.c_i = i$. By virtue of the tracking data structure we can show the following theorem:

Theorem 6.1 *When there are no object updates in the system and the system is in a consistent state, the object i is located in d hops where $d = dist(k.agent, i.agent)$*

In case object data structure is being updated, the $\mathbf{where(objecti, p)}$ message can reach a process r such that $r.c_i = NULL$, which reflects a previous location of the object i. In this case, the return value can be the earlier location or $NULL$ depending upon the application requirement.

Once the object is located, the reply to $\mathbf{where(objecti, p)}$ message is propogated through the $\mathbf{here(i, p)}$ message along the path to object k.

For any two objects i and j, recall that $dist(j.agent, p.agent)$ is propotional to the physical distance d between the two objects. Thus we have the following theorem.

Theorem 6.2 *The latency between an object k issuing a* $\mathbf{where(objectk, p)}$ *to* Trail *and receiving a* $\mathbf{here(k, p)}$ *from* Trail *is proportional to the distance between objects k and p.*

Stabilization Action Stabilizing actions for locating object i is implemented using a timeout at $k.agent$, the agent of the object 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.

6.3 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 the section on *Trunk*. We evaluate the performance of *Trail* udnder different scaling factors such as increasing number of objects, higher speed of objects and higher query frequency in terms of the reliability and latency offered to the application.

The clusters are of size 7. The backbone nodes operate at a power level by which they can reach 3 feet reliably while the non-backbone nodes operate at a power level sufficient to reach the backbone node. Note that 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 for upto 3 times.

We evaluate *Trail* with 2, 4, 6 and 10 objects, always in pairs. One object in each pair is the object issuing *where* query (say tracker) and the other object is the object being found (say trackee). In each of this scenario, we consider query frequency of 1 Hz, 0.5 Hz, $0.33hz$ and $0.25Hz$. The object speed affects the operation of *Trail* in terms of the rate at which *grow* and *clear* messages are generated. We consider 2 different object update rates, one in which objects generate an update every 1 second and other in which they generate update every 2 seconds. Considering that the object traces were collected

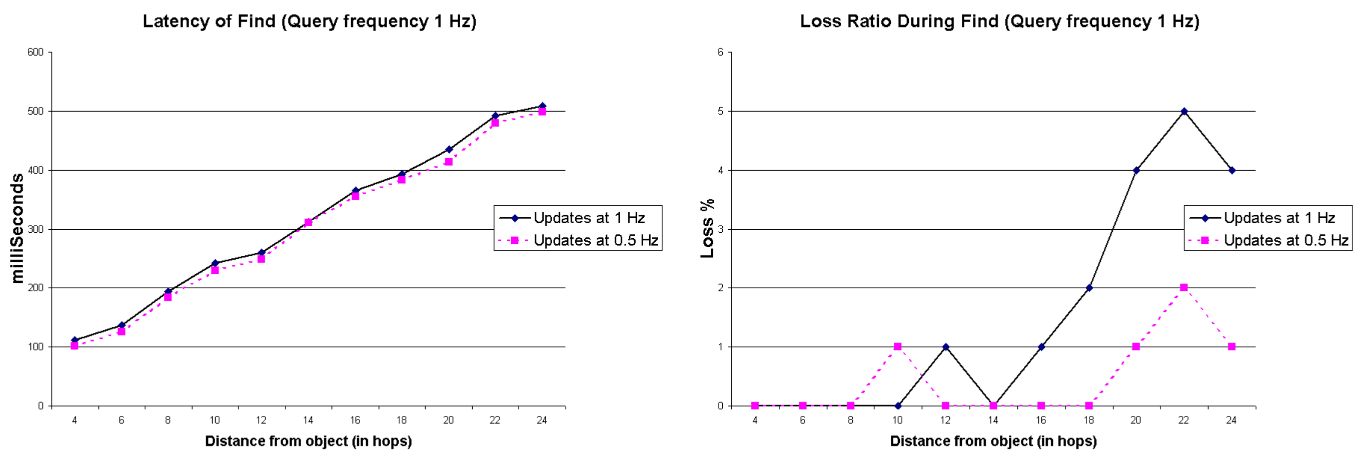


Figure 6. Latency and Reliability of Trail (2 Objects, Query frequency 1 Hz)

Figure 7. Latency and Reliability of Trail (2 Objects, Object Update Frequency 0.5 Hz)

with humans walking across the network acting as objects with average speed of aout $1m/s$, object update rates of $1Hz$ and $0.5Hz$ enable a tracking accuracy of $1m$ and $2m$ respectively. Note that each update can generate multiple *grow* and *clear* messages.

In the 4, 6 anfd 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 PEG application requirmeents 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.

The following figure shows the latency and loss for find operations when the objects are generating one update per second and one update per two seconds. The query frequency is 1 Hz. The latencies are measured at the node issuing the *where* and is the time elapsed between a *where* message being sent and a *here* message received. 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.

In the following figures, we show the reliability and latency of *Trail* with 4, 6 and 10 objects under different query frequencies with object update rate of 0.5 hz.

6.4 Using Trail for Pursuer Evader Tracking Application

In this subsection, we describe how trail is used by a multiple pursuer, multiple evader tracking application. In this case, the mobile objects are pursuers and evaders. We use *Trail* to support a pursuer evader tracking application called Intruder-Interceptor game [4]. In this application the objective of the interceptors is to maximize the distance of intruders from a valuable asset and the objective of intruders is vice-versa. One possible solution for this game is described in [4] where each interceptor is assigned to one intruder and there are critical requirements for the rate at which interceptors get data about their assigned intruders in order to meet the objective. The assignment of pursuers to evaders can be done optimally using a network service like *trunk* or *trail* could be used to assign intruders to nearest *free* interceptors. The design of the function in Trail to find nearest objects that satisfy certain criteria can be found ina related technical report and is not discussed here.

Once the interceptors have been assigned to intruders, the tracking can be done using *where(object i, object p)* function of Trail. Interceptors request information about intrusers at different frequencies depending upon their relative location and require information at higher frequency as they get closer. As seen in the evaluation section, the latency and reliability improves as objects get closer even at higher frequencies of queries.

7 Synchronous Trail

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 nodes 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 find and move operations are exactly like in Trail, but the network operates synchronously. 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 l clusters each. In alternate segments, message waves are schduled 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. 8.

We now analyze what is the minimum required l so that the backbone waves can proceed to the neighboring segments without incurring a delay at segment boundaries and still not interfere with the aggragation in the neighboring segment. Recall from our analysis in *Trunk* that the time required for a wave to traverse a segment of length l in both directions is $l + 4$ 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 comunication range of the neighboring segment so that the waves do not incur a wait at the boundary of the segments. 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 $l + 2 > n + 2$ or $l > n$ in order for the waves to not incur a wait at boundary segments.

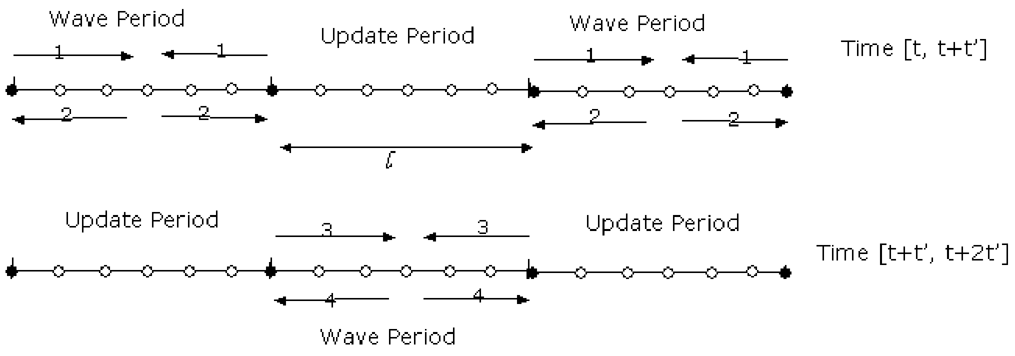


Figure 8. Synchronous Trail Operation

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 l.

Lemma 7.1 *The minumum required segment size for the backbone waves to not incur a wait at the segment boundaries is the smallest number l such that there are at most n' objects in l clusters and $n' < l$.*

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 atmost 3 objects within any region of 4 consecutive clusters, then we can satisfy the condition stated in the above lemma and l can be 4 or more.

We also note that in general, if $f(n')$ is the number of slots required to aggregate messages for n' objects from a non-backbone node to a backbone node, then $l + 2 > f(n')$.

We now analyze the energy and latency aspects of *Synchronous Trail.* The analysis for energy is similar to *Trunk.* Each backbone node is awake for 1 slot in the backbone wave time to listen to aggregated message from non-backbone node, 2 slots for listening to wave message from neighboring backbone and at most 2 slots to transmit to neighboring backbone node. The central backbone node in every segment is awake for 4 slots per 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 additonal types of delay, *aggregation delay*, *pickup delay* and *center delay.* The aggregation delay is the time required to send a message from non-backbone to a backbone node. This is a fixed delay of $l + 2$ time slots. The *pickup* *delay* is illustrated in the following figure. Depending on the position of a node within a segment, this delay is between 1 and l. The *center delay* is the delay introdced due to staggering of the messages at the center. This delay is 2 hops for every segment that the message passes through. Hence we can state the following lemma.

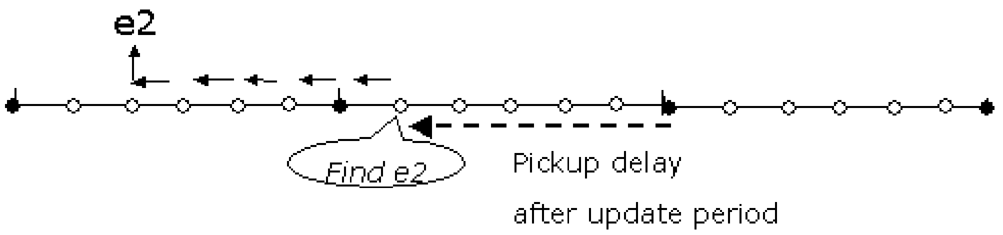


Figure 9. Pickup Delay after Aggregation to the Backbone

Lemma 7.2 *The latency for a query* **where(objecti, objectp)** *is given by Eq. 1, where l is the number of clusters per segment, s is the number of segments that the the* find *query passes through and dist(x,y) is the hop distance between nodes x and y.*

$$\begin{aligned} Latency &= 2 \times (dist(i.agent, p.agent) - 2 + (l+2) + l + 2s) && (1) \\ &= 2 \times (dist(i.agent, p.agent) + 2l + 2s) && (2) \end{aligned}$$

Thus, we see that there are energy 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.

We leave the experimental evaluation of the latency and reliability of *Synchronous Trail* for future work.

8 Conclusions and Future Work

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

Figure 10. Trunk: Network Service for Global Snapshots to Mobile Objects

```
**Protocol**    Trail at node j
**Var**
            $j.c_i$ : pointer to object i
            $j.p_i$ : pointer away from object i
            $j.detect_i$ : boolean
**Actions**
  **Track Update Actions**
  $\langle U_1 \rangle :: j.detect_i \longrightarrow$
                  $j.c_i = i;$
                  if  $(\neg j.center)$
                      $send_{j,j.unbr}$  $(grow(i);$
                      $j.p_i = j.unbr$
                  fi;
  []
  $\langle U_2 \rangle :: recv_{k,j}(grow(i)) \longrightarrow$
                  if  $(\neg j.c_i)$
                      $send_(j, j.c_i)$  $(clear(i);$
                      $j.c_i = k;$
                  []
                  $(j.c_i = \perp)$
                  $j.c_i = k;$
                  $send_{j,j.unbr}$  $(grow(i);$
                  $j.p_i = j.unbr$
  []
  $\langle U_3 \rangle :: recv_{k,j}(clear(i)) \longrightarrow$
                  if  $(\not j.c_i)$
                      $send_(j, j.c_i)$  $(clear(i);$
                      $j.c_i = \perp;$
                      $j.p_i = \perp;$
                  fi;
  []
  * * * * * * * * *
  **Stabilizing Actions for Track Updates**
  $\langle S_1 \rangle :: (j.p_i \neq \perp) \wedge (j.time - j.SendHbTimeout_i = HeartBeatTime) \longrightarrow$
                  $send_{j,j.p}$  $(hearbeat_i);$
                  $j.SendHbTimeout_i = HeartBeatTime$
  []
  $\langle S_2 \rangle :: recv_{k,j}(heartbeat_i) \longrightarrow$
                  $j.c_i = k;$
                  $j.p_i = j.unbr;$
                  $j.ReceiveHbTimeout_i = HeartBeatTime$
  []
  $\langle S_3 \rangle :: (j.c_i \neq i) \wedge (j.c_i \neq \perp) \wedge (j.time - j.ReceiveHbTimeout_i = HeartBeatTime) \longrightarrow$
                  $j.c_i = \perp;$  []
  * * * * * * * * *
  **Actions for Finding Object**
  $\langle F_1 \rangle :: recv_{k,j}(where(i,p)) \longrightarrow$
                  if  $(j.c_i \neq \perp \wedge j.c_i \neq i))$
                      $send_{j,j.c_i}$ (where(i,p))
                  []
                      $(j.c_i = i) \wedge (j.c_p \neq \perp)$
                      $send_{j,j.c_p}$ (here(state(p),i))
                  []
                      $(j.c_i = i) \wedge (j.c_p = \perp)$
                      $send_{j,j.unbr}$ (here(state(p),i))
                  []
                      $(j.c_i = \perp)$
                      $send_{j,j.unbr}$ (where(i,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 11. Trail: Network Service for Tracking Mobile Objects