Detection and Tracking of Dynamic Amorphous Events in Wireless Sensor Networks

Nicholas Hubbell
Lockheed Martin Corporation
Littleton, CO 80127, USA
nhubbell10@gmail.com

Qi Han
Department of Mathematical and Computer Sciences
Colorado School of Mines, Golden, CO 80401
qhan@mines.edu

***Abstract*—Wireless sensor networks may be deployed in many applications to detect and track events of interest. Events can be either point events with an exact location and constant shape, or region events which cover a large area and have dynamic shapes. While both types of events have received attention, no event detection and tracking protocol in existing wireless sensor network research is able to identify and track region events with dynamic shapes(NH-change shapes to identities), which arise when events are created or destroyed through splitting and merging. In this paper, we propose DRAGON, an event detection and tracking protocol which is able to handle all types of events including region events with dynamic identities. DRAGON employs two physics metaphors: *event center of mass*, to give an approximate location to the event; and *node momentum*, to guide the detection of event merges and splits. Both detailed theoretical analysis and extensive performance studies of DRAGON's properties demonstrate that DRAGON's execution is distributed among the sensor nodes, has low latency, is energy efficient, is able to run on a wide array of physical deployments, and has performance which scales well with event size, speed, and count.**

***Index Terms*—event tracking protocols; wireless sensor networks; energy efficiency**

I. Introduction

Wireless sensor networks have been considered very useful for event detection and tracking in various applications such as oil spill detection or ground water contaminant monitoring. The challenge here is to devise a method for the sensors to recognize, label and follow these events as they travel through the network. This identification and tracking capability forms a critical foundation for various higher level processing tasks such as predicting an event's evolution or conducting queries on the affected area. For instance, for some applications like monitoring the dispersion of fluids, classic numerical fluid transport models for fluid prediction are extremely computationally intensive and require hours to run to completion. In order to monitor events in real time, the model should be decomposed and computation should be distributed among the sensor nodes to exploit computational parallelism. By identifying and tracking each event in a distributed manner, one node for each identified event can be designated as an interface point for running the model.

Events are defined within two classes. The first class is "point events" that have a precise location associated and possess a static, well defined shape with a crisp boundary. The second class is "region events" that are less restrictive and can encompass the first. Region events are large amorphous phenomena to which a single location coordinate cannot apply. They are allowed highly dynamic shapes that shift with time. Moreover, such events may have "fuzzy" boundaries that are not easily seen and somewhat subjective.

The chief shortfall in current efforts as detailed in Section II is the universal assumption that events never combine into a large whole nor disintegrate into several smaller phenomena. Scenarios where this assumption does not hold are easily possible. Consider again the chemical spill as it diffuses below ground. If the fluid is pouring out from more than one site, the separate plumes may meet and mix together. In so doing they lose their individual shapes in a single large cloud. Conversely, changes in the medium through which it permeates may cause the fluid to follow a few preferred paths and break up into separate, smaller concentrations. In practice, keeping track of the dynamic expanding, shrinking, dividing, and merging of contaminant is essential to making treatment decisions.

Our goal is to design a protocol that is able to detect, label, and track wide area, amorphous region events with dynamic evolution. Events that exhibit this property are capable of dynamically splitting apart or merging together.The protocol is expected to run on a wide array of deployments, so the distribution of nodes is allowed to be arbitrary and no particular regular or stochastic node placement is assumed. Moreover, the field is permitted to be three-dimensional. We assume that nodes are location-aware. Localization can be done during deployment if GPS is not an option.Dealing with packet loss is considered a separate issue that has been well studied in the literature, so a fault tolerant technique can be applied and implemented as one of the underlying services for this detection and tracking protocol.

II. Related Work

The approaches to region event tracking in existing research may be categorized into statistical methods [1], topographical techniques [2]–[8], and edge detection algorithms [9]–[12]. A statistical method is presented in [1] for detecting and tracking generic homogeneous regions without the benefit of an *a priori* predicate to identify events. Instead, it uses a kernel density estimator to approximate the probability density function of the observations. It is suggested that the detection routine be rerun periodically to accommodate the scenario of any new regions or holes that evolve in the midst of tracking. Even so, there is not an elegant way to handle new detections and persistent tracking in the same moment.

An example of the topological and contour mapping technique is Iso-Map [6], which builds contour maps based solely on the reports collected from intelligently selected "isoline nodes" in the network. This approach is limited to a plane. Another technique [7] collects a time series of data maps from the network and detects complex events through matching the gathered data to spatio-temporal data patterns. Essentially the work provides a basic infrastructure and then outsources the problem solution to the user, instead of directly solving the event tracking problem. SASA [8] uses a hole detection algorithm to monitor the inner surface of tunnels, where sensor nodes may be displaced due to collapses of the tunnels. In our work, node positions do not change due to the evolution of the phenomena being observed.

In edge detection based region event tracking, the challenge is to devise a method for nodes to be identified as "edge nodes" that are near the boundary of a region and from that, calculate an approximate boundary for the region in question. Three methods guided by statistics, image processing techniques, and classifier technology are developed and compared in [9]. A novel method for edge detection of region events makes use of the duel-space principle [11], [12]. The algorithm is fundamentally centralized, but it can be distributed among backbone nodes in a two-tier architecture. This approach, like [9], does not accomplish event labeling, so it cannot be directly applied to our task.

Existing research on point event tracking includes various tracking protocols such as [13], [14] and DCTC [15], [16] and theoretical contributions [17]–[20]. One of the most notable contributions is DCTC [16]. It uses a "Dynamic Convoy Tree" protocol to accomplish both event tracking and communication structure maintenance. DCTC essentially forms and maintains a spanning tree over the nodes which sense the event, which is perhaps the most obvious and straightforward method of tracking events within the network. It is a "boiler-plate" solution. Moreover, many of the existing high level event tracking services either cite DCTC directly or at least assume a spanning tree structure like it as part of the middleware needed for their query support. Despite its clean conceptual elegance, DCTC is still unsuitable for our task since it assumes that point events will be persistent and distinct.

III. Foundation Concepts

The fundamental insights on which our approach is based are both unique and intuitive. Two central concepts define how fully generalized events are detected and tracked. They are the notions of *event center of mass* and *node momentum*.

A. Event Center of Mass

An event's center of mass delineates the event's location. It is a generalized position coordinate that serves as a reference point for critical operations such as detecting the occurrence of event splits and merges.

Definition 1. Mass Function. The mass value $m_{n,t}$ represents the "intensity" of the event readings taken by node n at time t. It is a bounded non-negative real number as a function of the sensor reading. The exact formula of the mass function is application specific. There is an implicit binary event detection predicate with $m = 0$ meaning no detection.

Definition 2. Event Center of Mass. An event's center of mass is the average position of all nodes sensing the event weighted by the mass values. Let e be the event in question. Let $N_e(t) \subseteq N$ be the set of sensor nodes that detect event e at time t. Let $M_e(t)$ be the total mass. Let η_n be the average of the number of distinct sensing regions of nearby nodes cover any given point within the sensing region of node n. Let the vector $\mathbf{l}_n = [x_n, y_n, z_n]$ be the three-dimensional position coordinates of node n. Then the center of mass of event e at time t denoted by $\mathbf{l}_M(e,t)$ is computed as follows.

$$\mathbf{l}_M(e,t) = \frac{1}{M_e(t)} \sum_{\forall n \in N_e(t)} \frac{m_{n,t}}{\eta_n} \mathbf{l}_n, \text{ where}$$

$$M_e(t) = \sum_{\forall n \in N_e(t)} \frac{m_{n,t}}{\eta_n}$$

The node density η_n is used to ensure that the event center of mass is not skewed towards areas of low mass because there happen to be more nodes present in the area. One possible method for obtaining η_n is to estimate the shape of the average sensing region for a given node. In a particular deployment, calculate the expected amount of multiple overlap in sensing regions for each node using numerical integration techniques.

B. Node Momentum

To keep track of event splits and merges, we use the following intuition as a guideline. If a high concentration of an event's readings is moving far off the event's center of mass, then that concentration should be recognized as an autonomous event. Conversely, if two separate events are so close that their reading concentrations are practically indistinguishable, then they should be folded into a single whole. These thoughts expose what information is required in order to detect splits and merges. First, we must consider both reading strength and that reading's distance from the event's center of mass in tandem. Second, in deciding not just *if* but *where* to split a single event, distance must be augmented with direction, thus requiring a vector quantity. We hence adapt another concept from the realm of physics, the concept of momentum.

Definition 3. Node Momentum. The momentum of node n with respect to event e at time t is denoted by $\mathbf{p}_n(e,t)$ which is computed as $\mathbf{p}_n(e,t) = m_{n,t}\Big(\mathbf{l}_n - \mathbf{l}_M(e,t)\Big)$.

A node's momentum is its position vector relative to the event center of mass, scaled by its own mass reading. In our context, there is no time difference involved and all quantities are set at time t.

An example of the use of momentum is shown in Figure 1. While momentum vectors are always on a line passing through

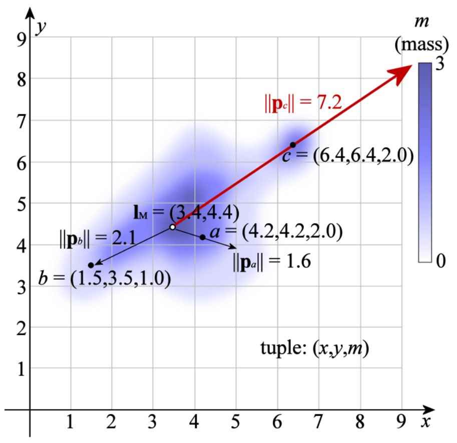


Fig. 1: An example of node momentum – The event's center of mass is indicated with the hollow dot at location $(3.4, 4.4)$. Also shown are three nodes with their locations, mass values, and arrows emanating from the center of mass as momentum vectors with their Euclidean magnitudes written alongside.

their nodes and the event's center of mass, they may not land right on their respective nodes. This is due to the scaling by the mass value. A momentum vector will only land on the node if the mass is 1, as is the case with node b in Figure 1. All of the other nodes have mass values over 1 and so their momenta overshoot their positions. In the figure, the momentum vectors are different shades and sizes to illustrate how they are used in a splitting event situation. Nodes a and b both have relatively small-magnitude momentum vectors shown as small black arrows. This is because they either have a high mass value with a short distance from $\mathbf{l}_M$, as is the case with node a, or a larger distance with a smaller mass, like node b. However, node c is a different scenario. It can be seen that c lies over a part of the event that is about to break off from the whole. Thus c has a high mass *and* a large distance, which together conspire to produce a momentum with a magnitude more than double that of the other nodes. This signals a split, so the momentum arrow is larger and a different shade. Therefore, we have the realization of the aforementioned intuition that if a high concentration of an event's readings are moving far off the event center, then that concentration should be recognized as an autonomous event. Merges are symmetric in logic.

C. Split and Merge Decision Predicates

The node momentum is the decision variable that controls whether two events should remain logically distinct or instead be folded into one entity. The possible outcomes of this decision control the event splitting and merging powers unique to our proposed solution. In a naive approach, a split occurs when a group of nodes have momenta with magnitudes above a certain value. This, however, is insufficient. In real situations, nodes will have high magnitude momenta naturally if an event is simply very large. A simple momentum threshold limits the size of events that can be detected. What needs to be determined instead is if a node's momentum is large relative to the event's overall size. To capture this insight, a cohesion threshold (T_C) will be placed on the *ratio* of a node's momentum magnitude to the average mass of a node sensing the event. The average mass per node is the total mass $M_e(t)$ divided by the number of nodes sensing the event $N_e(t)$. One of the problems with simply taking a ratio of momentum to event *total* mass is that the overall number of nodes could make the total mass grow abnormally, leaving a dependency on the deployment. A split occurs in event e if there is at least one node with the a momentum magnitude to average mass ratio that exceeds the threshold. This split predicate is formally stated as follows.

Definition 4. Event Split Predicate. Let T_C be a user defined *cohesion threshold* in units of distance. If the following predicate holds, then a split has occurred.

$$(\exists n \in N_e(t)) \left(\frac{\left\|\mathbf{p}_n(e,t)\right\|}{\frac{M_e(t)}{|N_e(t)|}} > T_C \geq 0 \right)$$

The predicate to decide a merge between events is symmetric to the one above. In this case, for each node in either of two events, their momentum vectors are calculated relative to the *combined* center of mass of both events taken together. This combined center of mass is called the *joint center of mass* and is formally defined as follows.

Definition 5. Event Pair Joint Center of Mass. For two events e_1 and e_2, their joint center of mass is denoted by $\mathbf{l}_M(e_1 \cup e_2, t)$ and is computed in the following manner.

$$\mathbf{l}_M(e_1 \cup e_2, t) = \frac{M_{e_1}(t)\mathbf{l}_M(e_1,t) + M_{e_2}(t)\mathbf{l}_M(e_2,t)}{M_{e_1}(t) + M_{e_2}(t)}$$

A node n's momentum relative to this combined center of mass is called its *joint momentum*.

Definition 6. Joint Node Momentum. A node n's joint momentum with respect to the pairwise combination of events e_1 and e_2 is denoted by $\mathbf{p}_n(e_1 \cup e_2, t)$ and is computed as $\mathbf{p}_n(e_1 \cup e_2, t) = m_{n,t}\Big(\mathbf{l}_n - \mathbf{l}_M(e_1 \cup e_2, t)\Big)$.

A merge occurs if all of the nodes in both events have a joint momentum to combined total mass ratio less than the cohesion threshold.

Definition 7. Event Merge Predicate. Let E be the set of all events in the network. A merge occurs between two events e_1 and e_2 if the following predicate holds.

$$(\exists e_1, e_2 \in E) \left((\forall n \in (N_{e_1} \cup N_{e_2})) \left(\frac{\left\|\mathbf{p}_n(e_1 \cup e_2, t)\right\|}{\frac{M_{e_1}(t)+M_{e_2}(t)}{|N_{e_1} \cup N_{e_2}|}} \leq T_C \right) \right)$$

If an event does *not* satisfy the split predicate, then it is considered to be *individually stable*. Similarly, if two events *do* satisfy the merge predicate, then they are considered to be *jointly stable*.

The values for sensor readings and node locations are only approximate in any real deployment. To avoid the split and merge decisions being highly sensitive to the noise, we could put the error tolerance on the cohesion threshold T_C.

IV. Algorithm Description

Using the foundation concepts, we have designed DRAGON, a distributed algorithm for detection and tracking of amorphous events with dynamic evolutions.

The Backbone Tree. We assume that the network is organized into clusters. Each cluster has exactly one node or "clusterhead" that serves as the local data sink for its respective cluster. DRAGON will be run on the clusterheads. If an event is entirely contained within one cluster, then that clusterhead can run DRAGON locally in a centralized manner. A foremost need is to allow clusterheads to take counsel with each other for cases where an event spans multiple clusters. Also, there is a need for global orchestration when deciding which existing events may be merged. To this end, we define the concept of the *backbone tree*. The links in the backbone tree are made from the inter-clusterhead links. Any clusterhead can serve as the root of the tree and as long as the tree is connected and contains all clusterheads it does not matter how the tree is formed. The underlying clustering protocol may choose to rotate clusterheads to balance energy consumption; it is also possible that the network consists of some resource sufficient nodes that serve as clusterheads.

Active Subtree Localization. To further illuminate the distributed nature of DRAGON, it is ideal that only those areas of the network which are actually tracking an event participate in the protocol to exploit locality and to save energy. To this end, the idea of the backbone tree is refined with the process of *active subtree localization*. This operation always precedes the main algorithms execution and happens as follows. First, the root sends out a small message to the entire tree asking each clusterhead if they are active or inactive. Whether a cluster is active or not depends on whether that cluster is a leaf node or an interior node in the backbone tree. A leaf node is active if and only if it has member nodes which are sensing an event; An interior node is active if and only if either the node itself is active or it has a child who reports being active. Leaf nodes respond first to the roots request. Active clusters will actually participate in DRAGON and inactive clusters will remain dormant through the run.

Action Triggers. Key phenomena for triggering the start of the algorithm are when nodes suddenly start to detect or cease to detect an event. A node n begins to detect an event if its mass value is greater than 0 for the first time. These nodes either spawn new events or join existing ones. A simple solution is to allow a newly detecting node to join an existing event if the mass functions are identical and the node preserves individual stability for the event in question. If there are no neighboring events, or all events would be made unstable with the addition of n, then n forms a new event. On the reverse side, the algorithm can be triggered when a node's mass drops to 0 and is removed from the event. In both the cases of adding and removing nodes, a minimum required node count can be used to avoid excessive calls, this minimum node count approach is called "node shift". Regular data collection periods are another possible trigger.

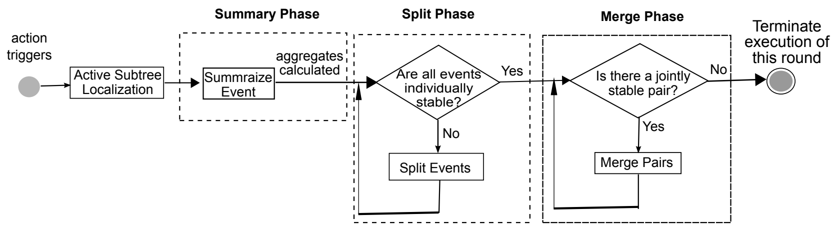
Algorithm Phases. The abilities and needs of DRAGON motivate three distinct phases of execution (Figure 2): Summary, Split, and Merge. As previously explained, the necessary decision predicates require three aggregates: center of mass, total mass, and node count. After active subtree localization, the Summary phase computes these aggregates for each event and distributes them to all active clusters. Information on all events is critical to deciding merges. The Split and Merge phases are charged with checking and performing event splits and merges respectively.

A subtle challenge exists in defining the relationship between the Split and Merge phases. To avoid endless thrashing between the two phases, we note that *if two events are jointly stable then the event which results from the union is itself individually stable*. Therefore, the product of a merge does not need to check for a split. This means that the merge phase can come after the split phase and there will be no cyclical dependency between the two phases, thus avoiding the thrashing problem. To provide the capability for multi-way splits and merges, we allow the normal two-way Split and Merge phases to run multiple iterations.

Intra-Phase Flow. The elegance of DRAGON lies in the fact all three phases of DRAGON follow a single unified pattern of state transitions, message passing, and computation. There is one algorithm template or "metaprogram" (Figure 3) running in all three phases with the details depending on the phase. The common form is informally a backbone tree-wide aggregation step followed by a multicast. The procedure involves three states: Start, Fusion, and Update.

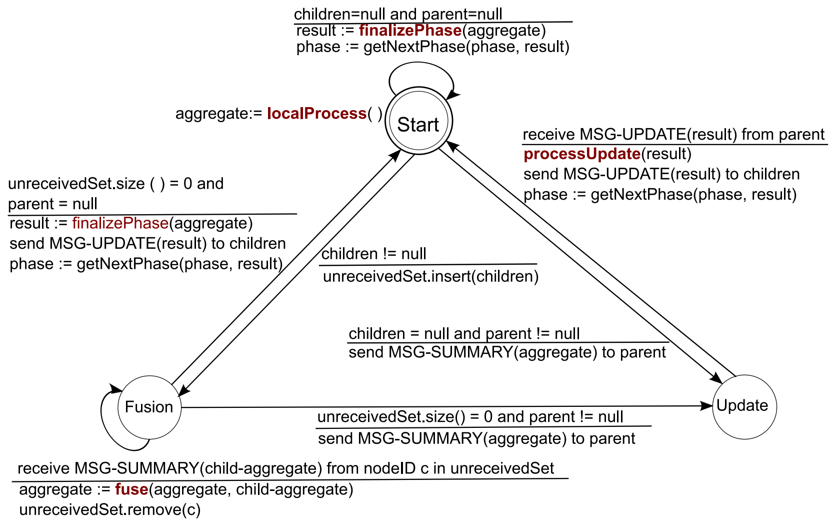
(1) The Start State. The program begins in the Start state where all clusterheads perform calculations limited only to those parts of those events within their respective clusters (via the `localProcess` routine). What transpires next depends solely on the clusterhead's status in the tree. a) If the clusterhead is the backbone tree root and also the only active clusterhead, then the phase can be finalized locally and the next phase can begin. b) If the clusterhead is a leaf node, then it can immediately send an `MSG-SUMMARY` message to its parent and go to the Update state. An `MSG-SUMMARY` message encapsulates the results of a clusterhead's local processing for all the events that it is tracking and is sent to that head's parent to be combined with the results from other clusters. c) If a clusterhead is either a relay node or the root node with active children, then it transitions to the Fusion state and waits for `MSG-SUMMARY` messages from its children.

(2) The Fusion State. The Fusion state is for a clusterhead (either a relay or the root) to aggregate its local results with those of its children. Upon receiving an `MSG-SUMMARY` message from a child node, the child's results are combined with the parent's results (via the `fuse` routine) for each event covered somewhere in the subtree. When all children have sent their results, the parent has complete results for its subtree. If the clusterhead is a relay node, then it sends those results to its parent via an `MSG-SUMMARY` message



action triggers
Active Subtree Localization
Summary Phase
Summraize Event
aggregates calculated
Split Phase
Are all events individually stable?
Yes
No
Split Events
Merge Phase
Is there a jointly stable pair?
No
Yes
Merge Pairs
Terminate execution of this round

Fig. 2: DRAGON's three phases: summary, split, and merge



children=null and parent=null
result := finalizePhase(aggregate)
phase := getNextPhase(phase, result)
aggregate:= localProcess( )
Start
receive MSG-UPDATE(result) from parent
processUpdate(result)
send MSG-UPDATE(result) to children
phase := getNextPhase(phase, result)
unreceivedSet.size ( ) = 0 and parent = null
result := finalizePhase(aggregate)
send MSG-UPDATE(result) to children
phase := getNextPhase(phase, result)
children != null
unreceivedSet.insert(children)
children = null and parent != null
send MSG-SUMMARY(aggregate) to parent
Fusion
Update
unreceivedSet.size() = 0 and parent != null
send MSG-SUMMARY(aggregate) to parent
receive MSG-SUMMARY(child-aggregate) from nodeID c in unreceivedSet
aggregate := fuse(aggregate, child-aggregate)
unreceivedSet.remove(c)

Fig. 3: DRAGON's metaprogram state machine shared by all three phases: summary, split, and merge. The details of the four subroutines (i.e., `localProcess`, `fuse`, `finalizePhase`, `processUpdate`) vary from phase to phase.

and transitions to the Update state. If instead the clusterhead is the tree root, upon receiving all children's results it now has complete information for all events and can perform final arbitration for the phase (via the `finalizePhase` routine) and processes the results itself (via the `processUpdate` routine). With the decision made, the root sends a multicast with an `MSG-UPDATE` message to all other active tree nodes to inform them of the outcome and moves directly from the Fusion state to the Start state for the next phase.

(3) The Update State. This state is for clusterheads that have sent their results to the root and are waiting for the root's multicast of the phase results. All nodes in the Update state upon receiving an `MSG-UPDATE` message from their parents preform the appropriate responses (via the `processUpdate` routine) before moving to the Start state and the next phase.

Details of DRAGON. We now discuss the details of the metaprogram subroutines — `localProcess`, `fuse`, `finalizePhase`, `processUpdate`— for each of the three algorithm phases.

(1) The Summary Phase. The `localProcess` routine executed at each clusterhead is to compute the center of mass, total mass and node count *for only those portions of events residing directly in that individual cluster*. The `fuse` routine takes the aggregates for local center of mass, local total mass and local node count for each event as computed by both the child and the parent nodes and combines them with an average and sum operations. The `finalizePhase` and `processUpdate` routines are involve sending and receiving the complete aggregates for all events in the network respectively. Then, the clusterheads' internal data structures are updated. A table of these aggregates is needed by every active cluster with entries for every event in the network. This is so split and merge calculations can be distributed over the tree and still be accurate.

(2) The Split Phase. The `localProcess` routine of this phase is responsible for recognizing splitting groups within the cluster in question. It begins by identifying all the nodes in event e whose momenta satisfy the split predicate and sorts them in decreasing order of node momentum magnitude. The algorithm repeatedly runs through these nodes, forming one split group per pass. For each node that satisfies the split predicate, the following *group formation predicate* is tested.

Definition 8. The Group Formation Predicate

Let there be a node n considered for inclusion into split group g. Let $M_g(t)$ be the total mass of all nodes already in split group g and define $\overline{\mathbf{p}}_g(e,t)$ to be the average momentum of all nodes in group g relative to event e's center of mass measured at time t. d is the distance. The quantity $K_g(t)$ is the number of nodes within split group g. T_S is the separation threshold used to control how far off a node's momentum vector may be from the avgMomentum of the group relative to group size. The group formation predicate is defined as follows: $\frac{d\left(\mathbf{p}_n(e,t),\overline{\mathbf{p}}_g(e,t)\right)}{\frac{M_g(t)}{K_g(t)}} < T_S$.

If this predicate is satisfied, then the node under consideration is added to the split group and the group's center of mass, total mass, and avgMomentum values are updated to reflect the addition. The node is finally removed from the list of nodes under consideration. Upon finishing a pass over all the nodes, the group being constructed is considered finished and the next group starts. This process continues until all the nodes being considered are added to a group.

The `fuse` routine in this phase is to scan the split groups acquired by the children and the parent nodes and fuse together groups that correspond to a single splitting phenomenon. The process of combining two groups is similar to the way that `localprocess` adds nodes to a group. The `finalizePhase` routine is tasked with selecting only one split group among potentially many to actually break off from the whole. The criterion we elect to use is to select the split group "farthest" from the main body of the event. The reference points employed for this comparison are center of mass of the split group and the center of mass of what remains of the original event should that group be split off. The `processUpdate` routine lets a clusterhead inspect the groups for the events that it is tracking to see if part of its cluster is included. If so, then for each relevant splitting event, a new event is recognized containing the nodes involved. Data structures for all events are updated accordingly.

(3) The Merge Phase. The `localProcess` routine arises directly from the test for joint stability between two events. The goal for the clusterheads overseeing portions of event e is to determine which events in the network satisfy the joint stability requirement with event e. The list of events that do is called a joint stability set $J(e,t)$. In the `fuse` routine, each individual cluster c_i has its own joint stability set for a given event e in its cluster, $J_{c_i}(e,t)$, but the sets of different clusters may disagree. An event e can only merge with events on which all clusters covering e agree. The parent node takes its own joint stability set and all of its children's joint stability sets for the same event e and computes the common intersection. As with the other fuse routines, events not shared by both parent and child are stored unchanged. The `finalizePhase` routine gathers all of the complete joint stability sets for all events and uses the table of event data to pick nearest jointly stable event pairs as merging pairs. The `processUpdate` routine requires clusters tracking the merging events to update their internal state by folding nodes from both events into one entity.

V. Theoretical Analysis

We here summarize theoretical results concerning the properties of DRAGON. Due to page limitations, only a proof sketch is provided. Detailed proof and analysis may be found in [21].

Theorem 1. DRAGON terminates.

The proof is done by showing that the Split and Merge phases will end after a finite number of iterations and there is no cyclical dependency between the two phases.

Theorem 2. DRAGON's decision predicates are invariant with respect to the range of the mass function m.

The theorem is proven by showing that DRAGON behaves identically if the mass function value is multiplied by a constant ρ such that $\rho \geq 0$. All of the algorithm's key decisions regarding the presence and nature of event splits and merges hinge on two ratios used in Definition 4 and Definition 7 and how the group formation predicate (Definition 8) is affected. The first is the ratio of node momentum to event average mass. The second is the ratio of inter-momentum distance to average mass. Both ratios are compared to the cohesion threshold T_C. Straightforward math derivation shows that the scaling factor ρ cancels out of the relevant ratios for all decision predicates of the algorithm. Therefore, the scaling of the mass function is invariant with respect to the decision predicates and the theorem is proven.

Theorem 3. DRAGON's execution is invariant to node placement, i.e., the decisions that DRAGON makes with respect to event splits and merges are not influenced by local differences in the density of the sensor deployment.

The proof begins with a lemma which shows that the aggregates of event center of mass and event total mass are invariant with respect to node placement. We then prove that DRAGON's execution as a whole is invariant to node placement by arguing that none of the major decision predicates have a dependence on node placement.

We have conducted detailed analysis of DRAGON's overhead in terms of time and message complexity. Details of the derivations are omitted for the sake of space. We will use C to denote the set of clusterheads throughout the network, C_E to denote the set of clusterheads covering events, N_E for the set of all nodes which can sense events, and E for the set of all events.

Time Complexity. The average case of time complexity:

$$\overline{T} = O\left(|E|\,|C_E|\,\lceil \log_b\left(|C_E|\right)\rceil + |E|\right)$$

DRAGON's execution time grows linearly with the number of events. It also grows $|C_E|\,\lceil \log_b\left(|C_E|\right)\rceil$ with the network coverage assuming a worst case which largely comes from assuming every active cluster contributes some split groups, and that these groups cannot be folded together across clusters. This is unlikely in practice. Our empirical results suggest time should grow linearly or better with event size and field coverage.

Message Complexity. The average case of message complexity concerns the number and size of messages transmitted during the algorithm's execution. We do not explicitly account for transmission power, since it is more dependent on both the underlying clustering algorithm, the physical deployment, and the radio model used.

$$\overline{M} = O\left(|E|\,|C_E|^2 + |N_E| + |C|\right)$$

Though $\overline{M}$ appears to scale quadratically with the number of active clusters, this assumes that there are split groups formed by every cluster for every event and that they cannot be aggregated together.

We conclude that DRAGON's costs scale linearly with the number of events and grow quadratically or better with respect to event coverage.

VI. Performance Evaluation

Event Modeling. It is impossible to cover all conceivable phenomena. In addition, we need to test our algorithm in various environments, so the size and number of events should be easily controllable. Therefore, we model events using basic geometric shapes. Region events are modeled with squares and point events have a circular sensing profile. We allow simple events to overlap, so we not only get splits and merges, but also we create much more complex and interesting event shapes by compounding simple geometry. Region events have only two reading levels, an outer band mass value of 0.5 and an inner plateau mass of 1.0. Point events have a linear drop off of mass readings which start at 1.0 at the point's location and fall radially down to 0.0 at the edge. Events start out a given run with a completely and uniformly random initial location on the 500 m by 500 m field. They also have a uniformly random initial direction of movement. The events move in a straight line, though they randomly change direction as necessary to avoid the edges of the field and, if merges are not allowed, each other. The speed of their movement remains constant at one of the pre-specified values.

Performance Metrics and Parameters. We measure the communication overhead with respect to energy consumption and execution time. All of our calculations are based on the TelosB mote platform [22]. The network is modeled as lossless and without collisions.

The other important metric is the notion of event detection and tracking accuracy that compares DRAGON's answers to an objective standard. In our simulations, the modules in charge of collecting statistics have direct knowledge of these underlying event shapes and evolution (the tracking algorithm being simulated does not), i.e., the ground truth. An accuracy measure must decide (a) whether DRAGON successfully detects all of the distinct events in the network and (b) whether each individual event is delineated correctly. The first measure is simply the *event count difference* between DRAGON's result and the ground truth. The second measure quantifies the correctness of an event's shape. We use a single metric called *event membership similarity* in a manner similar to the Jaccard Coefficient used in data mining [23]. It is defined as the ratio of correct matches to total nodes (i.e., the summation of correct matches, false positives, and false negatives).

Varying Parameters. We performed our simulation in our own modeling software inspired by TOSSIM [24], a widely used discrete event-driven simulator for wireless sensor networks. All nodes are deployed in a 500m by 500m field. Table I lists five independent variables along with each of their respective value ranges. Each node's transmission range is 20 meters.

Parameter	**Value Range**
Network Deployment (grid or random)	Sparse Distribution: ≈ 600 nodes Medium Distribution: ≈ 1200 nodes Dense Distribution: ≈ 2400 nodes
Event Width	75 m,150 m, 250 m
Event Speed	10 m/s, 20 m/s, 30 m/s, 40 m/s
Event Count	2, 4, 6, 8
Event Type	Point events that do not merge Point events that merge Region events that do not merge Region events that merge

TABLE I: Primary Variable Parameters

Comparison Algorithm. We chose the optimized DCTC [15] as the competing algorithm. This is because its overall operation is that of a spanning tree that reconfigures to cover the nodes which sense the event. As such, it represents a real-world example of the most basic, minimal effort "boiler-plate" solution possible for the event tracking problem. We also extended DCTC to R-DCTC to work with generic event shapes by subsuming the process of pruning old nodes and adding new ones in tree reconfiguration which assume no particular event shapes.

Experimental Results. Our first class of experiments deals with finding the right cohesion threshold (T_C) and separation threshold (T_S) for DRAGON while varying each independent parameter. Since accuracy is preeminent among our concerns, we optimize the thresholds only with respect to event count difference and event membership similarity. We have found, for a sparse network, $T_C = 0.8$ and $T_S = 0.9$ result in the best tracking accuracy when there are three events each moving at 20m/s; for a medium distribution network, the optimal $T_C = 1.3$ and $T_S = 1.2$; and for a dense distribution network, the optimal $T_C = 1.4$ and $T_S = 1.2$. We also found that the optimal cluster size is 2 (i.e., each member node is at most 2 hops from clusterhead) and the best action trigger is a node shift with threshold of 3. In these results (which are omitted due to space constraints but may be found in [21]), the curves change gradually and have a large plateau where performance is good. This means that the there are actually wide intervals of where the thresholds give good results. The parameters are not hypersensitive. It is not difficult to find the sweet spot for a given application. Also, the optimal thresholds do not change very much between deployments. This confirms the analysis that DRAGON's execution is invariant to node density.

Our second suite of experiments focuses on comparative testing between DRAGON and DCTC (for point events) or R-DCTC (for region events). We study the impact of deployment type, event size, event speed, event count, and event type. We only show the most interesting results here, and complete results may be found in [21]. In the following results, 95% confidence intervals are depicted and each data point is the average of 10 runs.

A. Impact of Event Size. We evaluate both protocols for different event sizes in a sparse network with 3 mergable region events moving at 20m/s. Figure 4a shows that DRAGON's event count difference is superb and R-DCTC is consis-

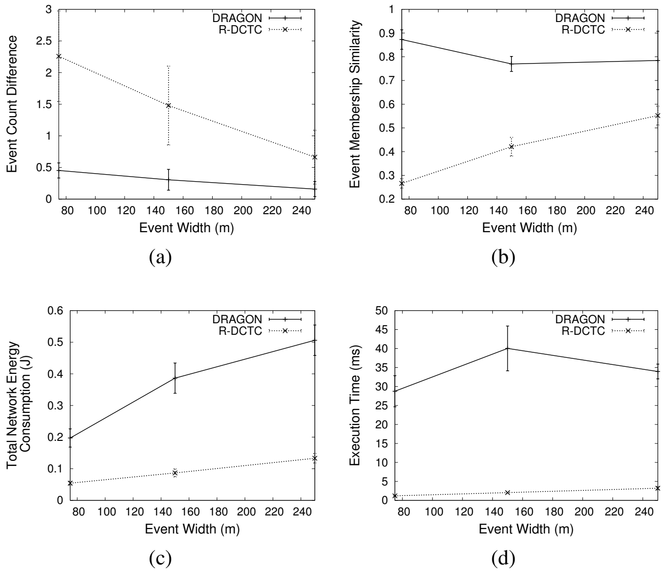


Fig. 4: Comparative performance vs. event size

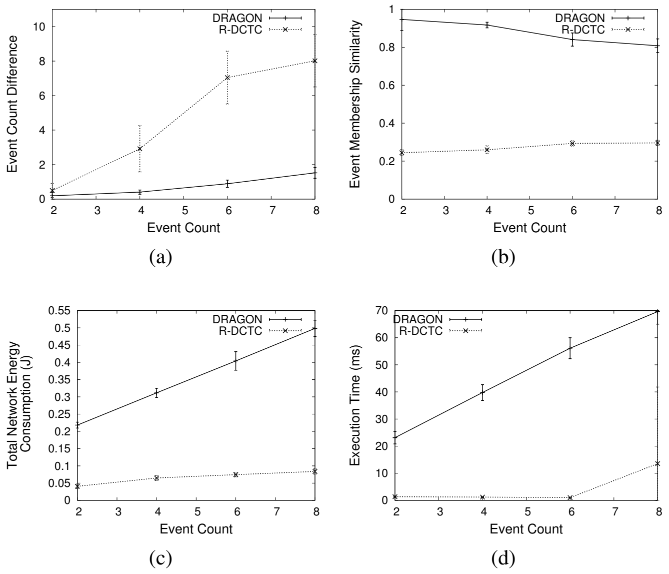


Fig. 5: Comparative performance vs. event count

tently wrong for various event sizes. Figure 4b indicates that DRAGON's event membership similarity hovers around 80%, but R-DCTC is never good. Moving on to Figures 4c and 4d, we must admit that DRAGON's costs are noticeably higher. Overall, DRAGON shows excellent accuracy, and though it has comparatively high costs in terms of time and energy, it scales fairly well. There is also another important conclusion. More importantly, we observe that DRAGON's cost in both time and energy consumption are somewhere between logarithmic and linear as event size increases.

B. Impact of Event Count. To evaluate both protocols for larger numbers of events, we consider mergable region events with width of 150m each moving at 20m/s in a sparse network. Figures 5a and 5b demonstrate that DRAGON shows consistently high accuracy which does not degrade much as the number of events increases. Meanwhile, R-DCTC has its worst accuracy ratings so far and results get consistently worse as the number of events goes up. The most fascinating results, however, are seen in Figures 5c and 5d. DRAGON has a very clear linear growth order for both energy consumption and time complexity as the number of events increases. Meanwhile, the energy consumption or execution time of R-DCTC has a very slight linear growth, but it is almost flat. This difference in asymptotic performance between the two protocols is not a problem though. Indeed it is expected. Because it handles splits and merges, DRAGON's computation and communication must explicitly consider the number of events in the network. The costs grow accordingly.

C. Impact of Event Speed. We evaluate both protocols for three events moving at various speeds in a sparse network with each event width of 150m. DRAGON has excellent accuracy with an event count difference less than 1 and a similarity of at least 80% as witnessed by Figures 6a and 6b. R-DCTC

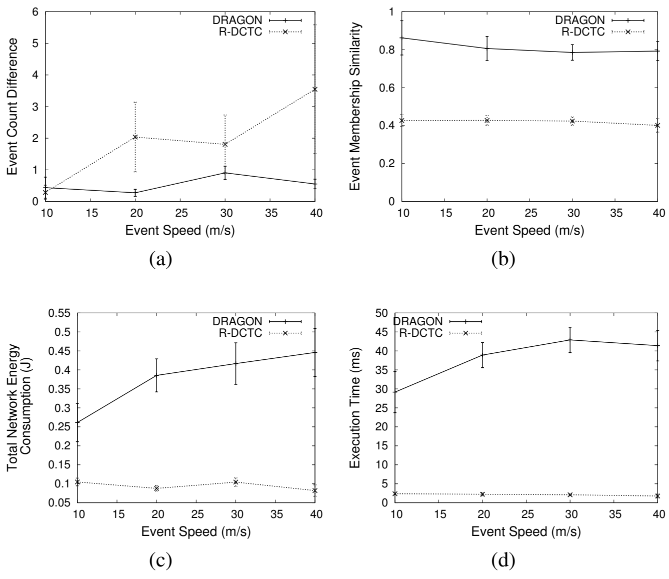


Fig. 6: Comparative performance vs. event speed

continues to falter, especially at 40 m/s. Figure 6c shows that DRAGON in general has a very slight linear increase in energy consumption as event speed grows. However, it is very minor and stays within a factor 3 or 4 of R-DCTC's cost, which is a very good result considering the extra work DRAGON involves. In Figure 6d we see that R-DCTC is clearly better with respect to execution time. However, DRAGON still shows good scalability.

We have also found that DRAGON shows excellent accuracy, competitive energy efficiency, and scalable time complexity with varying network density or topology. Results may be found in [21].

D. Impact of Event Type. To evaluate both protocols for all event types, be they point or region events, whether they come in contact with each other or not, we use a sparse network with 3 events with a width of 150m each moving at 20m/s. Figures 7a and 7b show again that DRAGON performs handsomely for point or region events irrespective of their mergability, while DCTC is very poor. Figures 7c and 7d show that DRAGON's energy costs are consistently within a constant factor of about 3 or 4 of DCTC's costs while the time complexity of DRAGON, although not comparable to DCTC, is still easily acceptable.

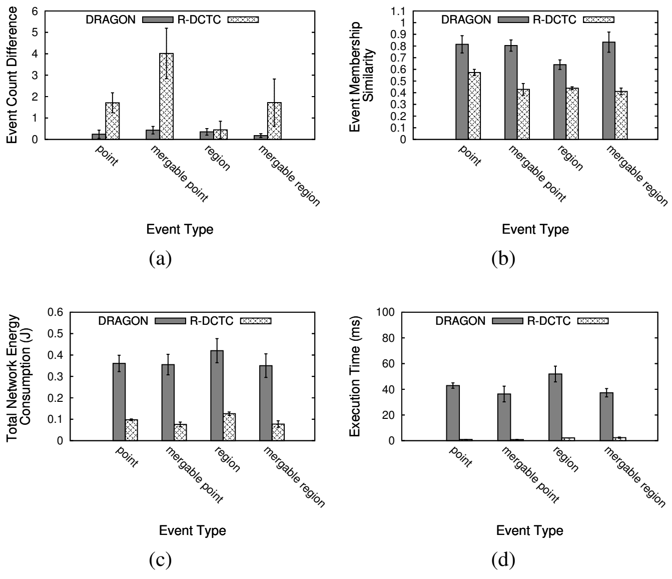


Fig. 7: Comparative performance vs. event type

Performance Summary. In summation, we draw the following conclusions. First, DRAGON solves the generalized event tracking problem well, but DCTC or R-DCTC does not. Second, DRAGON is stable and robust to a large variety of environments and problem sizes, and R-DCTC is fragile. Third, DRAGON does cost more energy than R-DCTC due to the nature of the expanded problem. Last, DRAGON cannot easily compete directly with R-DCTC in terms of time complexity. However, it would still be fast enough and scalable enough to keep up with events.

VII. Conclusions

In this work we have presented DRAGON, a general purpose event detection and tracking algorithm that is able to operate in the presence of event splits and merges. DRAGON has been shown to be highly accurate across a wide range of scenarios. It consistently finds the right number of events and outlines the right event shapes regardless of deployment type, and regardless of event size, speed, or count. DRAGON's energy efficiency scales well with problem size and complexity. The energy cost's order of growth is always shown to be linear or better with respect to the number of events. This cost is also consistently within a reasonable constant factor of the achievable optimum when varying other parameters like event size or network deployment. DRAGON's execution time is projected to be well within the constraints necessary to keep up with virtually any kind of event. Overall, DRAGON is promising for applications using wireless sensor networks for phenomena monitoring.

Acknowledgement

This work is supported in part by the NSF grant CNS-0915574.

References

[1] S. Subramaniam, V. Kalogeraki, and T. Palpanas, "Distributed real-time detection and tracking of homogeneous regions in sensor networks," in *RTSS*, 2006, pp. 401–411.
[2] C. Buragohain, S. Gandhi, J. Hershberger, and S. Suri, "Contour approximation in sensor networks," in *DCOSS*, 2006, pp. 356–371.
[3] S. Gandhi, J. Hershberger, and S. Suri, "Approximate isocontrours and spacial summaries for sensor networks," in *IPSN*, 2007, pp. 400–409.
[4] M. Singh and V. Prasanna, "Energy-efficient and fault-tolerant resolution of topographic queries in networked sensor systems," in *ICPADS*, 2006, pp. 271–280.
[5] W. Xue, Q. Luo, L. Chen, and Y. Liu, "Contour map matching for event detection in sensor networks," in *SIGMOD*, 2006, pp. 145–156.
[6] Y. Liu and M. Li, "Iso-map: Energy-efficient contour mapping in wireless sensor networks," in *ICDCS*, 2007, pp. 36–44.
[7] M. Li, Y. Liu, and L. Chen, "Non-threshold based event detection for 3d environment monitoring in sensor networks," in *ICDCS*, 2007.
[8] M. Li and Y. Liu, "Underground structure monitoring with wireless sensor networks," in *IPSN*, 2007.
[9] K. K. Chintalapudi and R. Govindan, "Localized edge detection in sensor fields," in *SNPA*, 2003, pp. 59–70.
[10] R. R. Brooks, P. Ramanathan, and A. M. Sayeed, "Distributed target classification and tracking in sensor networks," in *IEEE*, August 2003, pp. 1163–1171.
[11] J. Liu, P. Cheung, F. Zhao, and L. Guibas, "A dual-space approach to tracking and sensor management in wireless sensor networks," in *WSNA*, 2002, pp. 131–139.
[12] ——, "Apply geometric duality to energy efficient non-local phenomenon awareness using sensor networks," *IEEE Wireless Communication Magazine*, pp. 62–68, December 2004.
[13] J. Liu, J. Liu, J. Reich, P. Cheung, and F. Zhao, "Distributed group management for track initiation and maintenance in target localization applications," in *IPSN*, 2003, pp. 113–128.
[14] H. Yang and B. Sikdar, "A protocol for tracking mobile targets using sensor networks," in *SNPA*, 2003, pp. 71–81.
[15] W. Zhang and G. Cao, "Optimizing tree reconfiguration for mobile target tracking in sensor networks," in *INFOCOM*, 2004, pp. 2434–2445.
[16] ——, "DCTC: Dynamic convoy tree-based collaboration for mobile target tracking," *IEEE Trans. on Wireless Communications*, vol. 3, no. 5, pp. 1689–1701, 2004.
[17] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus, "Tracking a moving object with a binary sensor network," in *SenSys*, 2003, pp. 150–161.
[18] L. Lazos, R. Poovendran, and J. A. Ritcey, "Probabilistic detection of mobile targets in heterogeneous sensor networks," in *IPSN*, 2007, pp. 519–528.
[19] M. Nam, C. Lee, K. Kim, and M. Caccamo, "Time parameterized sensing task model for real-time tracking," in *RTSS*, 2005, pp. 245–255.
[20] N. Shrivastava, R. M. U. Madhow, and S. Suri, "Target tracking with binary proximity sensors: fundamental limits, minimal descriptions, and algorithms," in *Proceedings SenSys*, 2006, pp. 251–264.
[21] N. Hubbell, "Dragon: Detection and tracking of amorphous events with dynamic signatures in wireless sensor networks," Master's thesis, Colorado School of Mines, 2009, http://www.mines.edu/˜qhan/research/techrep/nhubbell-thesis.pdf.
[22] C. Solutions, "http://www.xbow.com, access date: January 2008."
[23] P. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison Wesley, 2006, ch. 2: Data.
[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *SenSys*, 2003, pp. 126–137.