DRAGON: Detection and Tracking of Dynamic Amorphous Events in Wireless Sensor Networks

Nicholas Hubbell and Qi Han, Member, IEEE

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 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.

1 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 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 precise locations and possess static, well-defined shapes with crisp boundaries. 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 2 is the universal assumption that events never combine into a large whole nor disintegrate into several smaller phenomena. Stated another way, all existing tracking solutions assume that while event shapes may be dynamic and nebulous, event signatures or identities are static and well defined. This implies that either events must remain distinct, never crossing or passing too close together to become indistinguishable to the algorithm, or if they do cross that they were identified prior and nothing new has formed. 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.

Problem statement. Our goal is to design a protocol that is able to detect and track wide area, amorphous region events with dynamic evolution. Events that exhibit this property are capable of dynamically splitting apart or merging together, thus meaning that events can be created or destroyed. 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 only require that the network forms a connected graph, the minimum necessary to assure communications. We assume that nodes are location aware. Localization can be done during deployment if GPS is not an option. 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 as well as communication relays for other clusters if clusterheads cannot communicate directly. Within each cluster, members organize themselves into a tree with the clusterhead as the root. The clusterheads collect (and optionally aggregate) the data from all members. Then, they forward the data to one or more base stations in the network by relaying it to the heads of neighboring clusters. This relay between the clusterheads can be accomplished either by high-powered, long-range direct communication or by establishing multihop communication paths using members of the clusters involved. 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 the protocol.

We have designed and evaluated DRAGON, an algorithm which can detect and track fully general, dynamically identifiable region events in an energy efficient manner in an arbitrary, connected, three-dimensional WSN deployment. To the best of our knowledge, DRAGON is the first work that have attained all of these goals.

The rest of the paper is organized as follows: In Section 2, we present related work. In Section 3, we present the concepts and intuitions that are the underpinnings of our approach. In Section 4, the design of DRAGON is given. In Section 5, we present the results of a formal analysis of DRAGON’s properties. In Section 6, the experimental evaluation is given.

2 RELATED WORK

The approaches to region event tracking in existing research may be categorized into statistical methods [25], topographical techniques [3], [5], [12], [13], [14], [23], [29], and edge detection algorithms [2], [4], [16], [17], and edge detection algorithms [2], [4], [16], [17]. A statistical method is presented in [25] 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 [14], which builds contour maps based solely on the reports collected from intelligently selected “isoline nodes” in the network. The approach is limited to a plane. Another technique [13] collects a time series of data maps from the network and detects complex events through matching the gathered data to spatiotemporal 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 [12] 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 [4]. A novel method for edge detection of region events makes use of the duel-space principle [16], [17]. The algorithm is fundamentally centralized. Yang and Fujita [31] identify several critical points in a given event region and periodically check the criticalness of these points, but the scheme can only work for an event whose shape remains convex.

Existing research on point event tracking includes various tracking protocols such as [18], [30] and Dynamic Convoy Tree Configuration (DCTC) [32], [33] and theoretical contributions [1], [10], [20], [22]. One of the most notable contributions is DCTC [32]. 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. Despite its clean conceptual elegance, DCTC is still unsuitable for our task since it assumes that point events will be persistent and distinct.

3 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.

3.1 Event Center of Mass
The concept of center of mass has been previously used to cooperatively control mobile sensor networks to track a moving target [9], clustering [27], and localization [15]. In DRAGON, we use an event’s center of mass to delineate 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. To define it, we first expound the idea of a mass function.

**Definition 1 (Mass Function).** The mass value \( m_n \) of an event \( n \) at time \( t \) is a bounded nonnegative 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.

Let \( e \) be the event in question. Let \( N_e \subseteq N \) be the set of sensor nodes that detect event \( e \). Let \( \eta_n \) be a measurement of the redundant sensing coverage in node \( n \)’s sensing region; that is, \( \eta_n \) is an average of how many distinct sensing regions of nearby nodes cover any given point within \( n \)’s
sensing region. The sensing regions considered include that of node $n$ itself. Let the vector $l_n = [x_n, y_n, z_n]$ be the three-dimensional position coordinates of node $n$.

**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, $N_e(t) \subseteq N$ be the set of sensor nodes that detect event $e$ at time $t$, $M_e(t)$ be the total mass. Let $\eta_n$ be a measurement of the redundant sensing coverage in node $n$’s sensing region; that is, $\eta_n$ is the average number of distinct sensing regions of nearby nodes that cover any given point within $n$’s sensing region. The sensing regions considered include that of node $n$ itself. Let the vector $l_n = [x_n, y_n, z_n]$ be the three-dimensional position coordinates of node $n$. Then, the center of mass of event $e$ at time $t$ denoted by $l_{CM}(e, t)$ is computed as follows:

$$l_{CM}(e, t) = \frac{1}{M_e(t)} \sum_{n \in N_e(t)} \frac{m_{n,t}}{\eta_n} l_n,$$

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

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

### 3.2 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 $p_n(e, t)$ which is computed as follows:

$$p_n(e, t) = m_{n,t}(l_n - l_{CM}(e, t)).$$

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 Fig. 1. The momenta vectors are always on a line passing through 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 Fig. 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 $l_{CM}$, 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 is moving far off the event center, then that concentration should be recognized as an autonomous event. Merges are symmetric in logic.

### 3.3 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:

$$\left( \exists n \in N_c(t) \left( \frac{\|p_n(e, t)\|}{M_n(t)/|N_v(t)|} > T_C \geq 0 \right) \right).$$

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 $I_M(e_1 \cup e_2, t)$ and is computed in the following manner:

$$I_M(e_1 \cup e_2, t) = \frac{M_{e_1}(t)I_M(e_1, t) + M_{e_2}(t)I_M(e_2, t)}{M_{e_1}(t) + M_{e_2}(t)}.$$  

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 $p_n(e_1 \cup e_2, t)$ and is computed according to the following formula:

$$p_n(e_1 \cup e_2, t) = m_n(t)I_n - I_M(e_1 \cup e_2, t).$$

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:

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

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 $T_C$. These inherent inaccuracies will affect the calculations of the split and merge predicates. This will in turn cause decisions to be highly sensitive to inputs that have some random fluctuations. There are several ways to alleviate this issue. The first option is to carry the basic tolerances through all of the calculations and augment calculated values with confidence estimates. The final values which are evaluated against the cohesion threshold $T_C$ will then be intervals with high and low extremes. To be conservative in this environment, an event would split only if the lowest possible value for the split predicate ratio of a given node is still above $T_C$. Similarly, two events would merge only if the highest possible value for the merge predicate ratio was still below $T_C$ for every node in either event. The second option is to put the error tolerances on the threshold $T_C$. The cohesion threshold would then essentially become an interval. Splits occur for momentum size to average mass ratios clearly above the high side of the interval and merges would happen if all nodes had rations below the low side of the interval, respectively. The interval could be estimated beforehand by prior knowledge of the location and sensing errors, the deployment, and the expected event size.

## 4 Design of DRAGON

Using the foundation concepts, we have designed DRAGON, a distributed algorithm for detection and tracking of amorphous events with dynamic evolutions. DRAGON is a powerful solution with a multifaceted approach. Fortunately, all of the various routines with their individual complexities are effortlessly amalgamated into a single whole by way of a unified routine structure.

In this section, first, we present some concepts that illuminate the distributed nature of our protocol. If an event is entirely contained within one cluster, then that clusters head 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 discuss the concept of the backbone tree to facilitate cooperation and to control DRAGONs execution throughout the network. On an related note, 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. Second, the action triggers that begin the algorithms execution will be exposed. Third, the relationships between the individual phases will be clearly defined. Last, a universal distributed template is outlined which all phases must follow.

### 4.1 The Backbone Tree

The links in the backbone tree are made from the interclusterhead links. Any clusterhead can serve as the root of the tree 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.

### 4.2 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. Active clusters will actually participate in DRAGON and inactive clusters will remain dormant through the run. Whether a cluster is active or not depends on whether that cluster is a leaf node or an interior node in the backbone tree. Leaf nodes respond first to the roots request. A leaf node is active if and only if it has member nodes which are sensing an event.

It must be mentioned that the clusterhead in question does not necessarily need to poll the entire cluster to determine this predicate. In practice, member nodes will take it upon themselves to inform the clusterhead of relevant changes to their readings via a sensor report. Such reports are sent in accordance with the given node’s duty cycle. A clusterhead will keep track of which nodes have positive detections in a table. That table is consulted when the active subtree localization request is received. A leaf clusterhead responds with a report containing this flag.

A clusterhead which is interior to the backbone tree waits to receive reports from each of its children before sending its own. An interior node cannot only consider itself when deciding if it is active. Although an interior node may not be actively tracking an event in its own cluster, it may need to act as a relay for a child node who is. It is clear then that the flag sent by an interior node in its report must represent the entire subtree rooted at that node. As such the flag is true if and only if either the interior node itself is active by the previous definition, or it has a child who reports as being active.

4.3 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.” Other possible triggers can be local mass shifts. Each cluster may maintain a local center of mass and total mass and call the algorithm when they change beyond certain amounts. Regular data collection periods are another possible trigger.

4.4 Algorithm Phases

The abilities and needs of the algorithm motivate three distinct phases of execution (Fig. 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, and they both come after the summary phase.

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.

Further, DRAGON must provide the capability for multiway splits and merges. If DRAGON attempts to test for multiway operations in one pass then there is the vexing problem of which way to do such operations. Consider the example of merging more than two events at a time. Each potential merge has a unique joint center of mass, this means that every possible merge combination between a set of events must be considered individually. Thus, to test in one pass for a multiway merge among events $e_1, e_2, e_3$, and $e_4$, the merge predicate given in Definition 7 must be tested for the combinations $\{e_1, e_2\}, \{e_1, e_3\}, \{e_1, e_4\}, \{e_2, e_3\}, \{e_1, e_2, e_3\}, \{e_1, e_2, e_4\}, \{e_1, e_3, e_4\}$, and $\{e_1, e_2, e_3, e_4\}$. An obvious combinatorial explosion in time and message complexity ensues, making the algorithm intractable. Therefore, we devise a simpler and more feasible scheme. To accomplish multiway operations, we allow the normal two-way Split and Merge phases to run multiple iterations before moving on.

4.4.1 Intraphase 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” (Fig. 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. The pseudocode for the complete process is given in Appendix A, which can be.

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

B. 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 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.

C. 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.

4.4.2 Splitting Groups
The details of the metaprogram subroutines—localProcess, fuse, finalizePhase, processUpdate—for each of the three algorithm phases are provided in Appendix B, available in the online supplemental material. We next elaborate on one of the most important tasks in the Split phase: how to recognize splitting groups within the cluster in question. This is accomplished by the localProcess routine. To be more specific, the routine first calculates node momenta for all the cluster members involved in a given event. Upon doing this, nodes are then identified that satisfy the split predicate given in Definition 4. These nodes are marked and then grouped together according to the similarity of their momenta. This procedure is done again separately for each event within the cluster. localProcess 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 distance between the individual node’s momentum and the group’s avgMomentum is calculated. This distance value d’s ratio to the split group’s mass is then compared against a new threshold $T_s$. This 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. This defines a new predicate formally presented as follows:

**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 \( \mathbf{p}_g(e,t) \) to be the average momentum of all nodes in group \( g \) relative to event \( e \)'s center of mass measured at time \( t \). Let \( d \) be 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:

\[
\frac{d(\mathbf{p}_n(e,t), \mathbf{p}_g(e,t))}{M_g(t) / K_g(t)} < T_S.
\]

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. It should be noted that the average momentum is calculated in a manner analogous to center of mass. There is an extra wrinkle in these computations; however, when averaging momenta, the mass values are normalized by the node density term \( \eta_n \) to avoid being skewed by the deployment.

### 4.4.3 Discussion

In DRAGON, we have delegated the problem of reliable network communication and robustness in the presence of node faults to the underlying clustering algorithm. While we never intend to require that DRAGON’s core logic handle faults directly, there are some extensions which could be implemented that would allow DRAGON to operate more gracefully in faulty situations. We will label such extensions by whether they are intracluster or intercluster measures. The major concern for intracluster fault tolerance would be to allow DRAGON to operate correctly in the presence of corrupted or missing values from cluster members. Some remedies include discarding lost data or replacing it with a guess based on past history and data from surrounding nodes. There are two major concerns for intercluster fault tolerance. The first is reliable message delivery between clusterheads, the second is recovery if a clusterhead fails. For the former case, third party nodes which overhear the backbone messages could cache and repeat the overhead messages in time if an acknowledgment from the destination is not forthcoming. The multiple sinks formulation mentioned earlier would also be a solution. For the latter case, an emergency backup clusterhead could be elected and request that its members and the backbone tree root send it the data it needs to continue.

### 5 Theoretical Analysis

We here summarize theoretical results concerning the properties of DRAGON. Detailed proof and analysis may be found in [8].

**Theorem 1.** DRAGON terminates.

**Theorem 2.** DRAGON’s decision predicates are invariant with respect to the range of the mass function \( m \). That is, the decisions made by DRAGON are the same regardless of the value of the mass function’s upper bound \( m_{\text{max}} \).

**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.

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.** DRAGON’s execution time is the time needed for all three phases, including the active subtree localization. The average case of time complexity

\[
T = O(|E||C_E|\log_b(|C_E|)) + |E|.
\]

DRAGON’s execution time grows linearly with the number of events. It also grows \(|C_E|\log_b(|C_E|)\) 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, including the messages used for active subtree localization. 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

\[
\mathbb{M} = O(|E||C_E|^2 + |N_E| + |C|).
\]

Though \( \mathbb{M} \) 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. That being the case, energy may, in practice, scale more linearly with the number of active clusters. Also, take note that although there are terms for the set of all active nodes and for the set of all clusters, active or inactive, that the messages involved with those terms are small, one-time expenditures per run.

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

6.1 Event Modeling

There are as many ways to model events as there are application areas where events occur; 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 prespecified values.

6.2 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 [24].

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 1) whether DRAGON successfully detects all of the distinct events in the network and 2) 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 [26]. It is defined as the ratio of correct matches to total nodes (i.e., the summation of correct matches, false positives, and false negatives).

The ground truth for event splits and merges is determined using the exact event centers known only to the event simulation. The simple event shapes are recursively merged if pairs of event centers draw very close. They are split in the reverse case.

6.3 Varying Parameters

We developed our simulation in our own software inspired by TOSSIM [11], a widely used discrete event-driven simulator for wireless sensor networks. Our simulations have perfect packet delivery success. There is no signal attenuation or message collisions. All nodes are deployed in a 500 m by 500 m field. To showcase DRAGON’s range and reliability faced with a whole range of different event configurations and behaviors and to also demonstrate that DRAGON is very flexible and performs well on different deployments, Table 1 lists five independent variables along with each of their respective value ranges. Each node’s transmission range is 20 meters.

6.4 Comparison Algorithm

We chose the optimized DCTC [33] 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. In addition, many of the existing high-level services such as EnviroSuite [19] and Regiment [21], [28] either cite DCTC directly or assume a spanning tree structure like it as part of their middleware for general query support. Therefore, DCTC holds some of the same aspirations for general extensibility as DRAGON. Overall, we pick DCTC for comparison because both its core abilities and long term goals are most similar to DRAGON’s. 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.

6.5 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 aim to find the best 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 20 m/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 \(T_S = 1.2\). Several important trends are apparent in the results (which are omitted due to space constraints but may be found in [8]). First, the metrics of event count difference and event membership similarity have a very strong and desirable correlation. At the optimal thresholds, the average event count difference is well below 1 for sparse and medium density networks, regardless of node placement. This suggests that DRAGON reliably discovers the right number

### Table 1: Primary Variable Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value Range</th>
</tr>
</thead>
</table>
| Network Deployment | Sparse Distribution: \(\approx 600\) nodes  
                    | Medium Distribution: \(\approx 1200\) nodes 
                    | Dense Distribution: \(\approx 2400\) 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  |
of events across an entire run. Second, event membership similarity peaks. This shows that DRAGON can perform very well in a wide range of environments. Third, $T_S$ has a negligible impact on performance. Fourth, in these results, 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. Also, the optimal thresholds do not change very much between deployments and they always hover in the same general area. This confirms the analysis that DRAGON’s execution is invariant to node density.

Among all the four action triggers (i.e., periodic, node shift, cluster center of mass shift, and cluster total mass shift) we implemented, we found that a node shift threshold of 3 is the best. Even more encouraging is that the VigilNet system [6], [7], a real-world testbed uses a node shift mechanism similar to ours with a threshold of 3 in order to decide if it has acquired a target. Given our results and the confirmation found in real-world experiments, we will use a 3-node shift action trigger for our comparative testing.

Cluster size not only affects accuracy, but also energy consumption and execution time. This is because fewer and larger clusters have a smaller backbone tree. On the flipside, data gathering within large clusters is much more costly. Our aim is to find the best possible tradeoff between intercluster and intracluster communication. By evaluating the impact of cluster size on event count difference, membership similarity, total network energy consumption, and execution time, we found that 2-hop clusters are the best.

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 event size, event speed, event count, event type, and deployment type. In the following results, 95 percent confidence intervals are depicted and each data point is the average of 10 runs:

6.5.1 Impact of Event Size

We evaluate both protocols for different event sizes in a sparse network with three mergable region events moving at 20 m/s. Fig. 4a shows that DRAGON’s event count difference is superb and R-DCTC is consistently wrong for various event sizes. Fig. 4b indicates that DRAGON’s event membership similarity hovers around 80 percent, but R-DCTC is never good. Moving on to Figs. 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. More importantly, we observe that DRAGON’s cost in both time and energy consumption are somewhere between logarithmic and linear as event size increases. The interesting fact here is that in Section 5, our complexity analysis predicted an average case time complexity $T$ that grew order $n \log(n)$ with respect to $|CE|$. We also predicted that the average case message complexity $M$ could grow quadratically in the number of covered clusters. Our results here are well within those analytical bounds.

6.5.2 Impact of Event Count

To evaluate both protocols for larger numbers of events, we consider mergable region events with width of 150 m each moving at 20 m/s in a sparse network. Figs. 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 Figs. 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. The growth shown by DRAGON for the time and energy metrics has very significant value beyond this, however. These results confirm our complexity analysis.
done in Section 5 where we showed that both the average case time complexity $T$ and the average case message complexity $M$ are both linear in the number of events $|E|$. That is exactly what we see in these tests. These results show that our analysis is extremely accurate.

### 6.5.3 Impact of Event Speed

We evaluate both protocols for three events moving at various speeds in a sparse network with each event width of 150 m. DRAGON has excellent accuracy with an event count difference less than 1 and a similarity of at least 80 percent as witnessed by Figs. 6a and 6b. R-DCTC continues to falter, especially at 40 m/s. Fig. 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 Fig. 6d, we see that R-DCTC is clearly better with respect to execution time. However, DRAGON still shows good scalability.

### 6.5.4 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 three events with a width of 150 m each moving at 20 m/s. Figs. 7a and 7b show again that DRAGON performs handsomely for point or region events irrespective of their mergability, while DCTC is very poor. Figs. 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.

### 6.5.5 Impact of Deployment Type

To evaluate both protocols on different networks, we use three mergable region events with a width of 150 m each moving at 20 m/s. Fig. 8 reveals that DRAGON’s accuracy is superb while R-DCTC’s is poor. When looking at Fig. 8b, DRAGON’s event membership similarity is consistently good, hovering around 80 percent. R-DCTC on the other hand is unsuccessful at about 40 percent. Next, we turn to Figs. 8c. As expected, DRAGON will inevitably cost more than an idealized, heavily optimized spanning tree-based protocol. That said, DRAGON consistently stays within a constant factor of 4 to 5 times what R-DCTC costs, maybe 6 for a couple of scenarios. In terms of scalability, the two are comparable and the scale of the difference is not prohibitively costly. Last, we turn to Fig. 8d. This metric is where the comparison between the two protocols breaks down. Because DCTC has a separate tree and execution context for each event, it is much faster than DRAGON. That being said, even if DRAGON’s raw execution time were multiplied by a significant factor to estimate the effect of packet loss, DRAGON’s time would still be easily low enough to follow changes in events. Overall, DRAGON can tackle a harder problem than R-DCTC while being more flexible about its underlying network. Moreover, it can do all of this while not incurring much more cost.
6.5.6 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. DCTC does not include logic of any kind for even checking for event splits and merges. Because of its simplified nature and minimal operation, DCTC and R-DCTC in practice represents a kind of idealized lower bound for time and energy costs with minimal concern for accuracy. DRAGON’s proximity to DCTC’s and R-DCTC’s costs should be viewed as its closeness to an “optimum” set of costs for the event tracking problem.

7 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 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. 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.

Acknowledgments

This project is supported in part by the US National Science Foundation (NSF) grant CNS-0915574.

References

Nicholas Hubbell received the BS degree in mathematics from the University of Colorado at Colorado Springs and the MS degree in computer science from Colorado School of Mines in 2010. He is currently a software engineer at Lockheed Martin Space Systems Company. His research interests include simulation and high performance computing.

Qi Han received the PhD degree in computer science from the University of California, Irvine in 2005. She is currently an associate professor in the Department of Electrical Engineering and Computer Science at the Colorado School of Mines. Her research interests include wireless sensor networks, mobile and pervasive computing, and network-centric data management. She is a member of the IEEE and the ACM.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.