Querying Moving Events in Wireless Sensor Networks

Giuseppe Amato[a], Stefano Chessa[a,b], Claudio Gennaro[a,*], Claudio Vairo[a,b]

[a]*Institute of Information Science and Technologies (ISTI) of the National Research Council (CNR), via Moruzzi 1, 56124 Pisa, Italy*
[b]*Department of Computer Science, University of Pisa, Largo Pontecorvo 2, 56127 Pisa, Italy*

Abstract

The detection and tracking of composite events in Wireless Sensor Networks often employ ad-hoc solutions that aim at detecting and tracking only specific types of events or use generic query languages that are not specifically built to manage events. We propose a new query language and an in-network query processing solution that enable the definition of queries to track and gather information from events, using wireless sensor networks. The proposed language provides clauses aimed at defining dynamic tracking tasks and the autonomous migration of the queries on the network, depending on the event mobility. We describe the query model and the language, discuss its implementation, and present the results of the comparison with a TinyDB-like approach. We show that our approach is scalable with event mobility and that it is more energy efficient than TinyDB-like approaches.

Keywords: Detection, tracking, wireless sensor networks, query language.

1. Introduction

The database paradigm has been proposed as an effective way of programming and managing a wireless sensor network [1, 2, 3, 4]. This methodology allows the use of SQL-like languages to query sensors in order to acquire, filter, and compare environmental parameters that can be measured by available transducers (e.g. temperature, pressure, magnetic field, vibrations, etc.).

This paper represents a step forward with respect to previous approaches in the field of event detection and tracking. We propose a paradigm where queries are not limited to express conditions and to return values acquired by transducers. Instead, our approach allows users to define composite events (e.g., "explosion", "gas leak", etc.) and to express queries to filter and fetch data from the events themselves. For instance, it is possible to request information on the size or the speed of a gas cloud. Clearly, the information on the event is synthesized by using information from the sensor transducers themselves. However, once events have been defined, programmers can abstract on event details and they can express queries directly on the high level events, rather than deciding which transducers, in which positions, and at which frequency the sensor should be queried.

The detection and tracking of events in Wireless Sensor Networks (WSN) [5], typically employ ad-hoc solutions that aim at detecting and tracking only specific types of events (for instance, related to vehicles moving along a road). With these approaches, event detection is obtained by analyzing, at application level, raw values acquired by transducers. Once an event is detected, the monitoring and tracking of its evolution is obtained by injecting (either user-driven or automatically) ad-hoc tasks in the network. Directed Diffusion [1] is the most notable example of this approach, in which the queries (called interests) are encoded in packets injected in all the sensors in the network.

More recently, the Directed Diffusion paradigm has evolved in other approaches [6], where the user can inject structured queries (with semantics inspired to query languages used in databases). Among these, we mention TinyDB [2], Cougar [3], and MaD-WiSe [4]. With these approaches, the user also specifies tasks for the management, filtering and processing of sensed data. However, these languages do not provide any support for tracking moving events.

*Corresponding author
Email addresses: giuseppe.amato@isti.cnr.it (Giuseppe Amato), ste@di.unipi.it (Stefano Chessa), claudio.gennaro@isti.cnr.it (Claudio Gennaro), vairo@di.unipi.it (Claudio Vairo)

Our proposal allows users to define event detection and tracking strategies that are injected in the network to instruct sensors to cooperatively and autonomously detect and track arbitrary user-defined events. A *Composite Event* is defined as a combination of raw sampled data, possibly heterogeneous, obtained cooperatively by a group of sensors in a given area of the network. Detection and tracking tasks migrate autonomously among sensors in order to follow event movements. We refer to this strategy as *Dynamic Tracking*. Using the proposed query language (called EQL: Event Query Language), it is possible to gather information directly from detected events during their evolution. Specifically, the main aspects of novelty of EQL are that the detection and tracking tasks are expressed directly on events rather than on sensor readings (consequently, queries address the events) and the concept of event chain that allows us to start automatically a tracking task when a chain of activating event occurs. Moreover, the query processing mechanism supporting EQL is entirely executed in the network by the sensors themselves. To the best of our knowledge, it is the first time that such an approach, where events are treated as a sort of "first-class citizens" for a query language, has been proposed in the WSN field.

A preliminary effort to define a query language for tracking of composite events in WSN is given in [7, 8]. In these works, we focused only on the description of the language and on the modeling of composite events, but these papers does not address aspects related to the implementation of the language and present only a very preliminary evaluation of the approach. In the present paper, we further extend this approach by proposing new meta-operators to manage query migrations and an implementation scheme. We also implement a MATLAB simulator and we evaluate the efficiency of our approach, in terms of energy consumption against a Centralized Query Approach (hereafter referred to as CQA) that models the behavior of query processing systems such as TinyDB or Mad-WiSe in event tracking. Specifically, in CQA the sensors acquire the data and send them to a sink, that, in turn, performs the detection and pilots the tracking of the event.

The rest of the paper is organized as follows. Section 2 presents related works; Section 3 presents the model of events used in EQL and its query language, and Section 4 describes the implementation issues. The analysis and comparison with CQA is discussed in Section 5, and Section 6 draws the conclusions.

2. Related Work

Kumar et al. [9] propose one of the first frameworks for the distributed detection of composite events. The framework exploits a publish-subscribe mechanism. An application subscribes to an event by specifying the area where the event is expected to occur, and the protocol builds an event-based data collection tree. To detect composite events, a counter is maintained for each atomic event part of the composite event (the counter keeps track of the number of sensors that can sense atomic events). Sensors are then added to the tree until the counter reaches a predefined threshold. In this framework, the degree of distributed computation is limited to data aggregation in the data collection task, while the actual processing is executed in the sink. Moreover, the framework does not support natively the tracking of the events and its implementation is left to the application developer.

Abadi et al. proposed REED [10], an approach that extends TinyDB [2] with mechanisms for joins and multi-predicate queries, which enables the detection of composite events in this system. However, differently from our approach, its event detection mechanism is centralized and it is not designed to manage natively event tracking.

The Timely Energy-efficient k-Watching Event Detection problem (TEKWED) is examined in [11]. The authors propose to perform the actual event detection in one sensor (called gateway), which is responsible for reaching a conclusion and notifying the users if an event happens. Any sensor in the WSN at any time can become the gateway, in order to balance the energy consumption of the sensors. The sensors are grouped in detection sets and only one detection set is active at any time in order to save energy. The gateway disseminates in the WSN the threshold to be checked to detect the event, and the sensors notify the gateway when the current sensed values are over the threshold. The gateway then computes the information coming from the sensors and determines whether the event occurred. As compared to [9], this approach moves the computation for event detection inside the WSN, but this approach is still not fully distributed as most of the effort is in the gateway.

Escalation [12] is a new approach for the detection of complex events in WSN. The authors define a complex event as a set of data points constituting a pattern that shows that something interesting or unusual is occurring in the underlying process monitored. In their proposal, they use SAX (Symbolic Aggregate Approximation) to convert raw real-valued sensor readings to symbolic representations, and then they use a distance metric for string comparison

to measure similarity as well as pattern-matching in order to detect complex events. Users can also look for similar events by submitting templates. A template is a set of real-valued readings that is translated into a string and submitted to the relevant motes. Then, the streams of sensor readings are transformed into strings and compared with the string representing the template. A small distance on this comparison means that a similar event has occurred.

In [13] the authors propose a cluster based event detection scheme in which events are reliably forwarded to a sink in the form of aggregated data packets. In this approach, the sink assigns a dynamically adaptable reliability factor to clusters, according to their size and event proximity such that the clusters closer to the event send packets to the sink more frequently. In order to reduce the energy consumption, the scheme includes an energy-level based cluster head (CH) selection that ensures that higher energy nodes remain CH for longer time. In addition, it also provides a mechanism for the CH to control the transmission rate of the sensors according to the assigned cluster reliability. This work takes into account the energy consumption of the sensors and increases the degree of distributed computation, but the sink is still deeply involved in all the event detection activities.

In [14] the authors investigate the detection performance of randomly deployed WSN. They address the problem of detecting mobile targets that move continuously. They distinguish among rational targets (that have knowledge of existing sensors) and irrational targets (that are not aware of sensors and do not plan their paths). The authors propose a mechanism to find critical positions to deploy additional sensors, so that the freedom of targets can be limited and the probability of target detection is increased. However, the paper is only focused on how add sensors to enhance the coverage.

The Scheduling for Composite Event Detection (SCED) problem is addressed in [15]. The authors propose a sensor scheduling that ensures both coverage and connectivity requirements, and maximizes the network lifetime. The work assumes that sensors are densely deployed and that they are equipped with multiple transducers to detect composite events. The proposed method is to switch in idle redundant sensors and assumes time divided in round, each of which has a initialization and a data collection task. The initialization phase selects a set of sensors that guarantee both coverage and connectivity requirements (the other sensors are set to idle). The active sensors perform sensing and data relaying in the data collection task. A limit of this approach is that it requires preliminary knowledge of the sensing area and of the WSN in order to guarantee coverage and connectivity. Although the approach of this work can be considered distributed it does not deal with the problem of event tracking.

Zoumboulakis and Roussos propose [16] a family of algorithms based on online symbolic conversion of sensor readings for Complex Event Detection (CED) in WSN. The proposed CED algorithms have fixed execution cost and modest resource requirements, and they can perform exact, approximate, non-parametric, multiple and probabilistic complex event detection.

More recently, also the fuzzy logic has been applied to the event detection in WSN. FED [17] is a fuzzy event detection model that benefits from fuzzy variables to measure the intensity and the occurrence of detected events. FED uses fuzzy rules to define composite events to enhance handling uncertainty. FED also provides a node-level knowledge abstraction, which offers flexibility in applying heterogeneous sensors. The model is also applicable to a clustered network for distributed event detection. In [18] the authors focus on improving the definition of the event to be detected in order to increase the detection accuracy. In particular, observe that using precise values to specify event thresholds cannot adequately handle the often imprecise sensor readings. They use fuzzy values instead of crisp ones to define the event to detect, thus improving the accuracy of event detection. Although the work does not explicitly state it, this approach can be considered distributed.

All the works presented so far suffer from the same problem of not explicitly considering event tracking. Moreover, we are interested in *dynamic tracking* approaches, i.e., in which the query can automatically migrate in order to follow the moving event.

Several approaches address the tracking of moving objects in WSN, one of the first approaches is ZebraNet [19]. In this work, the sensors operate as a peer-to-peer network to compute local information about the moving objects. This information is then stored inside the WSN and collected periodically by a mobile sink. As in ZebraNet latency is not a requirement, the data on moving events may be retrieved after a long time since their acquisition, and this approach is hence not suitable for real-time tracking.

In [20] a prediction-based, distributed tracking algorithm is proposed. The algorithm uses a cluster-based architecture for scalability and robustness. Given a target to track, the proposed algorithm provides a distributed mechanism for locally determining the optimal set of sensors suitable for the task. Only the selected sensors are activated, thus minimizing the energy spent. Additionally, the protocol uses a predictive mechanism to alert the cluster heads about

the approaching targets. Based on this prediction, the cluster heads activate the most appropriate sensors for the task immediately before the arrival of the target. A distributed prediction-based approach is also used in [21], where the cluster organization of the network is exploited to track multiple moving objects. These approaches work well if the tracked events do not change their direction too often.

EnviroTrack [22] is an embedded tracking application for WSN. It adopts a data centric programming paradigm called attributed-based naming through "context labels", where the routing and addressing are based on the content of the requested data rather than the identity of the target sensor node. he attribute-based naming is applied by associating user-defined entities (context label) to real physical targets. With this network abstraction layer, the programmer declares the environmental characteristics that define the context label of the object to be tracked. Based on this, all sensors that sense the same declared characteristics (object) are aggregated to track that physical target. This approach does not propose a complete query language but rather a basic language with structures aimed at detection and tracking. In fact, differently from our approach, this language does not have the concept of Detecting Area and Detection Region, and more important, it misses the concept of alert the sensors nearby the area of the event. This latter concept is essential in our approach to guarantee the query migration, which, as observed in [23] is not guaranteed in EnviroTrack in all conditions.

In [24] the authors address the problem of tracking moving objects in a WSN by considering the physical network topology to build a logical tracking tree for the collection of tracking data. The tree is optimized in order to minimize the communication cost for updating the tree and the communication cost for executing the query.

RARE [25] proposes two algorithms for reducing the cost of target tracking in a statically clusterized WSN. The first algorithm reduces the number of sensors participating in tracking by excluding the sensors too far from the target object. The second reduces redundant information by identifying overlapping sensors.

CODA [26] is an algorithm for the detection and the tracking of continuous phenomena (such as fires, gas clouds, etc.) based on a hybrid static/dynamic clustering technique. A static backbone comprising a designated number of static clusters is constructed during the initial network deployment stage. Upon detection of a continuous phenomenon, the cluster heads of the backbone pilot the creation of dynamic clusters by using the information acquired by the sensors at the boundaries of the phenomenon, thus reducing the amount of communications between the sensors. The dynamic clusters are used to track the phenomena and to acquire data from them. This approach is not completely distributed and autonomous, since it relies on the sink for evaluating the boundary of the continuous object currently tracked.

Another approach [27] attempts at reducing the number of environmental readings and, as a consequence, the amount of data delivered outside the network. The proposed solution takes infrequent snapshots of the area and uses a low-quality target-tracking algorithm to maintain object identities. The target-tracking algorithm is leader-based: the node that is considered to have the best reading for the moving object coordinates the tracking. The leadership is passed from a sensor to another (a neighbor of the previous leader) by exploiting probabilistic estimation of the direction of the moving object. Periodically (exactly when the snapshot period elapses), the leader collects and aggregates the readings of its neighbors to achieve a high-quality belief on the target moving object, and it sends this information to the sink. This approach has a good performance with limited-mobility objects.

In [28] the authors present a novel framework for time-critical event generation in WSN that includes tools to model intruder detection events, as well as fire and gas propagation scenarios. The different models developed are integrated into an application optimized for ns-2 compatibility. They also provide a front-end to simplify the interactions with the user and to allow visualization of the different events generated. In [29] they use the framework developed in [28] to perform a comprehensive analysis of the performance of IEEE 802.15.4 based WSNs at supporting time-critical applications. In particular, they measure the accuracy and the delay introduced by gas and fire monitoring processes.

In [30] the authors propose a prediction based tracking technique using sequential patterns (PTSPs) designed to achieve significant reductions in the energy dissipated by WSN while maintaining acceptable missing rate levels.

The paper in [31] presents an efficient intelligent collaborative event query (ICEQ) algorithm to detect the event early and provide monitoring information and event query timely in WSNs. Ahn and Kim [32] propose a novel context detection mechanism in Wireless Sensor Networks, called PROCON in which they define the chain of events as a set of interrelated events with logical and timing relations among them. These latter two works, however, do not focus on the issues of query moving objects and track objects.

We refer the reader to [33] for a comprehensive survey and comparison of in-network querying and tracking

services for SWNs.

All the works presented above either address only a particular task in detecting events or tracking objects in WSN, or they are focused on a specific applicative scenario. On the other hand, we propose a comprehensive system that enables one to both detect a user-defined composite event and to automatically track it and collect interesting data from the event without any further intervention of the user.

Table 1 summarizes related work and compares it with our approach (EQL). For the sake of clarity, below, we give a summary of the aspects taken into account in the table. *Query language*, it includes approaches that propose a complete language for defining and tracking events. *Dynamic tracking*, it comprises approaches that in which the query can automatically migrate in order to follow the moving event. *Fully distributed*, approaches in which the sensor nodes cooperatively and autonomously detect and track user-defined events, without the aid of a node of coordination. *Event Tracking*, it includes approaches that propose techniques for tracking events in addition to strictly detected them. *Event chains*, it allows the developer the definition of an event that depends on a previously defined event. *Composite Events*, it includes approaches that an event condition to be expressed over a combination of raw sampled data obtained cooperatively by a group of sensors.

	Query language	Dynamic Tracking	Fully Distributed	Event Tracking	Event Chains	Composite Events
EQL (our approach)	√	√	√	√	√	√
DCL+COMiS [9]	√					√
REED [10]						√
TEKWED [11]						√
Escalation [12]			√			√
Kumar et al [13]		√				√
Zhou et al [14]						
Yang et al [15]			√			√
Zoumboulakis et al [16]			√			√
FED [17]			√			√
Kapitanova et al [18]		√				√
ZebraNet [19]		√		√		
Yang et al [21]		√	√	√		
Chong et al [21]		√	√	√		
EnviroTrack [22]		√	√	√		√
Lin et al [24]				√		
RARE [25]				√		
CODA [26]				√		
Tanin et al [27]		√	√	√		
Lino et al [28, 29]				√		
Raja et al [30]	√			√		
Zhu [31]			√			
PROCON [32]					√	

Table 1: Comparison of related work.

3. The Event Query Language

3.1. Events as Sources of Data

An event in the physical environment can be recognized by the occurrence of a combination of values measured by appropriate transducers. For example, an "explosion" can be characterized by a sudden peak of vibration, noise and pressure, and a subsequent increase of heat. The detection of an event can be modeled by defining a condition on the values measured by a specific set of sensors, installed in a specific area, as in the example shown in Table 2.

The occurrence of an event can be the consequence of another event. For instance, we can suppose that, after an explosion, a "gas cloud leak" event also occurs. As before, the event is detected by checking the occurrence of a

Explosion:	(`Accelerometer` > tA) AND (`Noise` > tN) AND (`Pressure` > tP)

Table 2: Example of an explosion event description. tA, tN and tP are given threshold values for, respectively, accelerometer, noise and pressure measurements.

SELECT	Position, Speed
FROM	GasCloud

Table 3: The query requests the position and the event speed GasCloud.

combination of measured values. However, given that the event depends on another event, the condition should be checked just after the occurrence of previous event. In this way, it is possible to define and take control of chains of related events.

The gas cloud example also suggests that there are events that evolve after they first occur. The gas cloud can move in the environment, can expand, or change density, etc. Therefore, in many cases, it is also useful to track the event in the space and to monitor its evolution.

Once an event has been detected and is being tracked, we may be interested in obtaining information about the event itself. For instance, once we detect that a gas cloud has been produced we may be interested in its position, its speed, its density (in case the gas cloud is moving or expanding), etc. In this work, we treat events as fist-class citizens and we express queries that use directly events as data sources. A very simple example of a possible query on the gas cloud event is given in Table 3.

Clearly, the information about the event can also be obtained by analyzing and acquiring data from sensors installed in the area where the event occurred. However, as the example in Table 3 suggests, we aim at providing users with a higher level of abstraction, so that users can express queries directly on events, rather than specific transducers in the WSN. With our approach, the user has just to decide which information should be obtained from the event. All details, related to activation of sensors, and strategies to detect and track the event, are specified once for all when the event is first defined, and are transparently managed by the event query processor at query time.

In order to deal with events according to the scenario above, we need to define a query language and in-network query processing strategies that allows users to (i) define events and data that can be read by events, (ii) define and process event detection tasks, (iii) define and process tracking tasks, and (iv) express and process queries that have events as data sources.

In the rest of the section, we address these points by defining the Event Query Language (EQL), a query language for querying and tracking events.

3.2. Declarative Language for Event Detection, Tracking, and Querying

The Event Query Language (EQL) allows defining events in terms of conditions on values measured by transducers and values returned by the event once detected. It supports the definition of rules for detecting the event and for tracking it during its evolution. It also allows expressing queries on events that gather data from them, and monitor their evolution.

The language reflects these aspects by providing users with the possibility of defining four different types of statements:

- *Event Statement* - conditions to recognize events and values returned by the event
- *Detection Statement* - rules specifying how and where to detect an event
- *Tracking Statement* - rules specifying how to track an event
- *Query Statement* - syntax for expressing queries on events

In the rest of the section, we provide the syntax and analyze each statement in details.

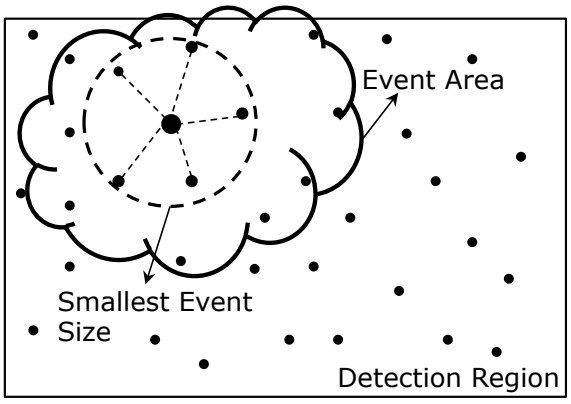


Figure 1: DETECTION REGION - The external rectangle represents the Detection Region where the event is monitored, the dashed circle represents the minimum expected size of the event (the Smallest Event Size that in this case is 1 hop) related to the sensor represented with the big black spot, and the cloud represents the actual area covered by the event when it occurs (the Event Area).

3.3. Event Statement

In order to define a new event we need to specify the conditions on the environmental parameters, used to recognize the event, and the attributes that can be read from the event after it is detected. In addition, we also need to specify the minimum expected size of the area that will be covered by the event, when it happens. We call this the *Smallest Event Size*. The Smallest Event Size also determines the minimum expected amount of contiguous sensors that are covered by an event when it occurs (see Figure 1). The decision about the occurrence of an event is taken by using the information acquired by all sensors in the radius defined by the Smallest Event Size. For instance, in order to detect an explosion, we can specify that the average computed on the values measured by a group of sensors, covering an area of the specified size, should be above a specified threshold. Isolated sensors individually measuring high values, do not detect the event.

Table 4 reports the syntax of the Event Statement. Tables 5 and 6 give an example of the definition of explosion and gas cloud events, respectively.

The `SIZE` clause specifies the extent of the Smallest Event Size in terms of number of hops between nodes. If we specify size n, the event will be detected when the values measured by a group of sensors, having pairwise hop-distance at least n, contribute to get a value above the threshold.

For instance, in the examples in Tables 5 and 6 the clause `SIZE` specifies that the Smallest Event Size is respectively 2 and 3 hops (if sensors are aware of their position, size can be expressed in Euclidean distance). This means that the event will be detected when there is a group of nodes, having pairwise hop distance as specified in clause `SIZE`, for which the condition expressed in the `WHERE` clause is satisfied.

The `AS` clause specifies the attributes that can be read from the event. Given that these values are computed by using the values acquired by all sensors in the radius defined by the Smallest Event Size, the returned values are obtained as aggregates. For example, in Table 5, the values returned by the explosion event are the noise, the average acceleration, and average pressure computed in an area of size 2 hops. In Table 6 the gas cloud is characterized by the average light, the average temperature, and the average level of oxygen in an area of size 3 hops.

The `WHERE` clause specifies the condition that should be checked to determine that the event occurs. Values defined in the `AS` clause can be used here.

```
eventSpecification::=
 DEFINE EVENT   <eventName>
 SIZE:          <len>
 AS:            <aggregate_list>
 WHERE:         <condition_list>
```

Table 4: The Event Statement

```
DEFINE EVENT  Explosion
SIZE:         2hops
AS:           Avg(Accelerometer) as accelExplAvg,
              Min(Noise) as noiseAll,
              Avg(Pressure) as pressAvg
WHERE:        accelExplAvg > 80 AND noiseAll > 30 AND
              pressureAvg > 90
```

Table 5: The Explosion definition.

```
DEFINE EVENT  GasCloud
SIZE:         3hops
AS:           Avg(Light) as lightGasAvg,
              Avg(Temperature) as tempAvg,
              Avg(Oxygen) as oxygenAvg
WHERE:        lightGasAvg < 50 AND tempAvg > 40 AND
              oxygenAvg < 60
```

Table 6: The GasCloud definition.

It should be observed that, for the sake of simplicity, the clause WHERE in the Tables 5 and 6 uses simple rules based on thresholds to detect the events. However, in general, the accurate detection of an event requires more complex data-fusion functions. Since the actual method of data fusion for event detection is out of the scope of this work, we refer the interested reader to papers [34, 35, 36].

3.4. Detection Statement

The Detection Statement defines the rules that will be used by the nodes of the WSN to perform the detection task of an event as defined by the Event Statement.

In several cases, it is not necessary to monitor the occurrence of an event in the whole environment covered by the WSN. Rather, only nodes deployed on some specific critical regions may execute this detection task. After an event occurs, the contribution of the neighbors of the sensors that detected the event is necessary to track its movements. For this reason, the neighbor sensors must be alerted to perform a monitoring task of the event. We call the critical region of the environment, where an event is initially monitored, the *Detection Region*. For instance, just nodes close to the gas tank, need to execute the detection of the explosion event. The remaining nodes will be alerted, if needed, in order to monitor the evolution of the event.

The Detection Statement specifies the Detection Region, the sampling rate the event conditions should be checked and other events that the current event might depend from.

The Detection Region identifies the set of nodes of the WSN where the occurrence of the event is initially monitored. All nodes in this area are instructed to perform the detection task [37]. When an event occurs, a subset of nodes of the Detection Region in an area of Smallest Event Size (as specified in the Event Statement), cooperate to detect the event (see Figure 1).

The sampling rate expresses the frequency at which the sensors must activate the transducers to sample the environment. This is necessary because some events can be monitored at low frequency (for example the rising or falling of the tide) while other events should be monitored at high frequency (for example an explosion), since their appearance can be very fast.

The syntax of the Detection Statement is shown in Table 7.

The `ON REGION` clause defines the Detection Region where the detection of the event should be initially executed. It can be expressed in different ways: by means of geographical coordinates, by a list of sensors identifiers that are known belonging to the region, by all the sensors of the network, or by an event name. In the latter case, the Detection Region of the event being defined corresponds to the set of nodes that actually contribute to detect an occurrence of the named event. This area is usually larger than the area defined by Smallest Event Size of the named event (see

```
detectionSpecification::=
 DEFINE DETECTION for   <eventName_list>
 ON REGION:             <area> | <id_list> | <all> |
                        <eventName>
 EVERY:                 <rate>
 [TIMEOUT:              <duration>]
```

Table 7: The Detection Statement

Figure 1). With this option, it is possible to define *Chains of Events*, where the definition of an event depends on a previously defined event.

```
  DEFINE DETECTION for  Explosion
  ON REGION:            DangerousZone
  EVERY:                500
  TIMEOUT:              30d
```

Table 8: The detection definition for the Explosion event

```
  DEFINE DETECTION for  GasCloud
  ON REGION:            Explosion
  EVERY:                1000
```

Table 9: The detection definition for the GasCloud event that depends from the Explosion event

Tables 8 and 9 show an example of the detection for the *Explosion* event and for the *GasCloud* event. In the example, the GasCould event depends from the Explosion event. This means that the detection task for the GasCould event starts after an explosion, using the area where the explosion occurred as Detection Region.

The `EVERY` clause sets the sampling rate of the event for acquiring data and evaluating the boolean expression. The `TIMEOUT` clause is optional and specifies how long the detection task has to be executed.

3.5. Tracking Statement

When an event occurs (and consequently the condition of the Event Statement evaluates to true) the detection task ends and the tracking task begins. The tracking task monitors the evolution of the event in time and space.

The event can evolve in multiple ways: it may just move maintaining its shape and its size (for example an intruder in a building, a car moving along a street, etc.), and/or it may change its physical properties (for example the gas cloud may move according to atmosphere conditions and it may change its shape and size).

The Tracking Statement specifies how the event should be tracked and the sampling rate for acquiring the data related to the event during tracking. The tracking task is executed by the sensors currently involved in the event. As we discuss in Section 4, one of the activities executed during the tracking task by nodes is to alert their neighbors (even outside the Alert Region) to start checking the detection conditions of the event being tracked. If alerted nodes detect the event, then they start executing the tracking as well. Elsewhere, they quit after a time-out.

Table 10 reports the Tracking Statement syntax and Table 11 reports the Tracking Statement for the GasCloud event.

```
trackingSpecification::=
 DEFINE TRACKING for   <eventName_list>
 EVOLUTION:            <alert_extension>
 EVERY:                <rate>
 TIMEOUT:              <sleepTime>
```

Table 10: The Tracking Statement

```
DEFINE TRACKING for  GasCloud
EVOLUTION:           1hop
EVERY:               1000
TIMEOUT:             5m
```

Table 11: The Tracking Statement for the GasCloud

The EVOLUTION clause specifies the area where the tracked event can be detected during its evolution. It represents the area where the neighbors of nodes executing the tracking task should be alerted to check the detecting condition of the event being tracked. This area is generally larger than the area where the event occurred, and in the case of Table 11, it is expressed in number of hops.

The EVERY clause specifies the sampling rate for the tracking task, and the TIMEOUT clause specifies the expiration time of the tracking.

3.6. Query Statement

The Query Statement is used to gather and process information related to an event. This statement is similar to a standard SQL query and uses events rather than tables in the FROM clause.

```
dataSpecification::=
 SELECT   <attribute_list>
 FROM     <eventName>
 WHERE    <condition_list>
```

Table 12: The Query Statement

```
SELECT  Position, Speed, oxygenAvg
FROM    GasCloud
WHERE   oxygenAvg < 50
```

Table 13: The Query Statement for the Gas Cloud example.

Table 12 reports the syntax of the Event Statement.

The SELECT clause specifies the list of attributes to be reported to the user. In general, these are properties of the tracked event among those defined in the Event Statement. We also suppose that standard attributes, like position, speed, and direction can be expressed as well. The FROM clause specifies the event to be used as source of information. Of course, data will start coming from an event just after it occurs. The WHERE clause specifies a list of conditions that have to be satisfied on the properties of the event.

Table 13 contains a query on the GasCloud event. The example requests the position and the speed of the gas cloud only when the average value of the oxygen is below 50.

4. Event Query Processing

In this section, we describe in more details how EQL queries are processed. EQL query execution is factorized three tasks: *detection* task, *dynamic tracking* task and *data collection* task.

4.1. Detection

When a Query Statement is submitted, the nodes included in the Detection Region specified in the query start executing the detection task. The pseudo-code of the detection task executed by all nodes of the Detection Region is reported in Algorithm 1. The detection task proceeds through the following steps: acquires the needed data (as defined in the Event Statement) from local transducers (1) and broadcasts them to neighbors reachable up to S hops (2). Then, in (3), it receives data acquired by its neighbors and, in (4), it evaluates the aggregate operators and the

Algorithm 1 DetectionTask (eventName)

Require: eventName

```
1: acquire data from transducers
2: broadcast acquired data up to S hops
3: receive acquired data from neighbors
4: whereClause = CheckCondition(received + local data)
5: if whereClause is FALSE then
6:     go to 1
7: else
8:     activeSensorList = EventAreaDefinition(localNode)
9:     TrackingTask(eventName, activeSensorList)
10:    GetDependentEvent(eventName, activeSensorList)
11:    go to 1
12: end if
```

Algorithm 2 EventAreaDefinition(localNode)

Require: localNode

```
1: start(Timer)
2: localActiveSensorList = neighbours + thisNode
3: broadcast localActiveSensorList up to S hops
4: receive remoteActiveSensorList from neighbors
5: updated = updateList(localActiveSensorList, remoteActiveSensorList)
6: if updated is true then
7:     go to 3
8: else if not expired(Timer) then
9:     go to 4
10: else
11:    return activeSensorList
12: end if
```

condition to check if the event occurred. If the event did not occur (5), then it repeats the previous steps. Elsewhere, the set of sensors, which contributed to detect the event, are collected by means of the EventAreaDefinition procedure (8). After this, it launches the tracking task (9), and checks if there are other events depending from this (10). In the latter case, their detection task is started as well.

The set of sensors that detect an event may cover an area larger than the one defined by the Smallest Event Size. We call *Event Area* the area actually covered by the set of sensors that contributed to detect the event. The pseudo-code of the procedure for the collection of the nodes involved in the event and the definition of the Event Area is shown in Algorithm 2. All nodes involved in an event execute this procedure locally. The aim is to provide all of them with the list of all nodes that detected the event. In order to avoid the procedure to run for a long time, a timer is set to stop the execution (1), and the list of nodes is initialized with the node executing the procedure (2). This information is broadcast to the neighbors reachable up to S hops (3). Then, in (4), it receives the list of nodes collected by its neighbors. The list of nodes collected is updated by making the union of the previous list and the list received by the neighbors (5). If the new list is different from the old one (6), then it jumps to (3) again. Elsewhere, if the timer has not expired (8), it jumps to (4). Eventually, the list of the collected nodes is returned (11). Note that with this procedure all nodes have the full list of nodes that participated in the detection of the event, so all nodes are aware of the Event Area.

4.2. Dynamic Tracking

Once the tracking task is triggered by the detection task, the nodes involved in the event start executing it cooperatively. In this way, they can contribute to track and collect data from the detected event. The pseudo-code of the

Algorithm 3 TrackingTask(eventName, activeSensorList)

Require: eventName, activeSensorList
1: leader=leaderElection(activeSensorList)
2: tree=treeBuilder(leader,activeSensorList)
3: **if** thisNode is boundary(tree) **then**
4: broadcast alert(eventName, leader, tree, timeout) up to S hops
5: **end if**
6: **if not** checkInclusion(eventName) and expired(timeout) **then**
7: return
8: **else**
9: activeSensorList=updateActiveSensorList(leader, tree)
10: go to 1
11: **end if**

tracking task is sketched in Algorithm 3.

Nodes executing the tracking task elect their leader (1), and then they build a routing and data collection tree (2). Nodes that are at the border of the event (that is, leaf nodes of the tree) alert their neighbors (lines 3 and 4), reachable up to S hops, to check if they also detect the event. Note that these nodes might currently be out of the Event Area, and they can become part of it if they detect the event. At this point nodes can check if they still see the event. If they no longer see the event and if the timer has expired, they terminate the tracking task (lines 6 and 7). Elsewhere, they update the list of nodes involved the event and restart the tracking loop (lines 8–10). Note that after the first execution of the tracking task, the tree is just updated, rather than created from scratch, by adding/removing new/old nodes. Moreover, the leader is updated only when needed, that is when it is not covered by the event anymore.

The tracking task pseudo-code makes use of the following procedures:

- leaderElection
- treeBuilder
- alert
- checkInclusion
- updateActiveSensorList

At the first iteration of the tracking task, the *leaderElection* procedure elects a leader according to distributed algorithms available in literature [38], [39] (the actual algorithm is not relevant to the purposes of this work, it can be based on the sensors' identifiers or it can be a more sophisticated one). In all the subsequent iterations, the new leader is not elected by means of a distributed algorithm (that is expensive in terms of energy consumption), rather a simple protocol that exploits the current tree is used. This protocol requires that the current leader knows the diameter, in number of hops, of the Event Area. This information can be easily obtained when the acquired data are sent back to the leader through the tree. When the current leader exits from the Event Area, it sends a direct message across the tree with a counter equals to the half of the diameter of the Event Area. Each node receiving this message decreases the counter until it reaches 0. The node that receives the message with counter 0 will be the new leader, and it will be located, approximately, in the middle of the Event Area. In order to ensure that only one node will be the leader, each node receiving this message (including the current leader that generates it) sends the message to the node with the minimum ID among its children.

If two or more events of the same type occur in two disjointed areas of the Detection Region, different detection tasks are executed. In this case different Event Areas are defined for the events detected and, correspondingly, different leaders. On the other hand, if the Event Areas of two or more identical events overlap, a unique Event Area, spanning the whole space covered by the detected events, is defined. In this case, a single leader is elected and a single data collection tree is built.

Name	Definition
Detection Region	Area where the event is monitored
Smallest Event Size	Minimum expected size of the event
Event Area	Area actually covered by the event
Active sensors	Sensors involved in the event
Passive sensors	Sensors not involved in the event
Active Boundary sensors	Active sensors that have at least one passive neighbor
Passive Boundary sensors	Passive sensors that have at least one active neighbor

Table 14: Glossary of the definitions related to the events.

The *treeBuilder* procedure builds a data collection tree spanning the whole Event Area. The leader coordinates this task, which is executed by all the other sensors in the Event Area. Each sensor receives a proper message containing the identifier of the sender, sets the sender as its parent on the tree and forwards the message with its own identifier. The tree is used to forward to the leader the interested data acquired from the event. This technique is quite common in literature [40], [2].

The *alert* procedure is executed by nodes at the border of the Event Area (leaf nodes of the data collection tree) to request to their neighbors to check if they also detect the event. Since we cannot predict the movement of the event, the node in the boundary notify the alert to nodes reachable within the number of hops specified by the `EVOLUTION` clause of the Tracking Statement. We call *active boundary nodes* the nodes that are at the boundary of the Event Area. We call *passive boundary nodes* those that have been alerted (since they are not part of the Event Area, until they detect the event). Alerted sensors are included to the data collection tree and they are notified about the leader (see Figure 2). Passive boundary nodes then start acquiring the data needed to detect the event according to the sampling period specified by the `EVERY` clause, and for a total time specified by the `TIMEOUT` clause of the Tracking Statement. If during this time an alerted sensor detects the event, it notifies the other sensors in the Event Area about the detection, so that the Event Area can be updated. Moreover, it executes the tracking task and starts acquiring the requested data from the event.

The *checkInclusion* procedure is executed by the sensors to check if the event does not involve them anymore. Each sensor periodically checks if the conditions of the tracked event (specified in the `AS` and `WHERE` clauses of the Event Statement) are still satisfied (the period is the same of the sampling of the requested data). When a sensor x does not detect the tracked event for a specified period and if it does not have at least one active neighbor, then x is removed from the data collection tree.

The *updateActiveSensorList* procedure updates the list of sensors involved in the event, as consequence of the addition of new sensors, or the removal of sensors. This procedure basically collects all nodes of the tree that have detected the event.

Figure 2 represents the data collection tree built for the gas cloud example. The tree is rooted in the leader sensor that is represented as the rounded circle. The figure also shows the alert notifications (the dotted arrows) sent by the active boundary sensors (the empty squares in figure) to the passive boundary sensors (the triangles). Table 14 summarizes the definitions used in the paper.

4.3. Data Collection

The detection and the tracking tasks have the purpose of preparing the network to the acquisition of data related to the tracked event. In particular, the detection task defines the set of active sensors, while the tracking task builds and maintains the data collection tree involving the active sensors.

Data collection is executed in the tracking task, after the data collection tree is created. It is performed by all the active sensors and it has the purpose of collecting the relevant data about the event, as specified in the Query Statement, and to send them back to the user. To this purpose, this task exploits the data collection tree created, and continuously updated, by the tracking task. In particular, the active sensors acquire, and possibly filter, data from the event and transmit them to the leader through the data collection tree. Data arriving at the root of the tree are processed by using techniques for in-network distributed WSN query processing already existing in literature. These issues are, for instance, extensively addressed in systems such as MaD-WiSe [4] or TinyDB [3]. The data resulting

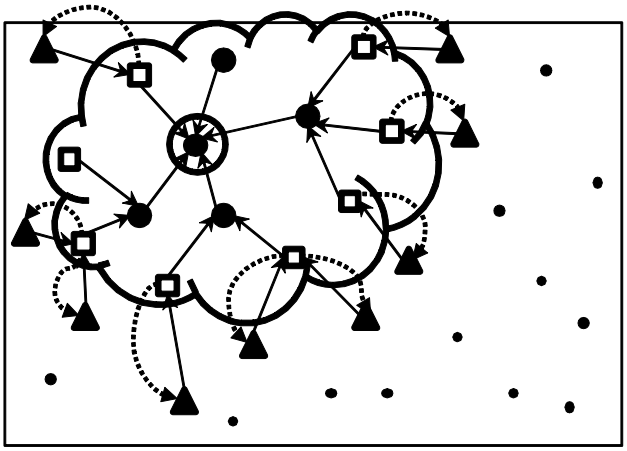


Figure 2: TRACKING PHASE - Representation of data collection tree built inside the Event Area. The tree is rooted in the leader sensor, represented as the rounded circle. We also show the updating operation: the Active Boundary sensors (the empty squares) alert the Passive Boundary sensors (the triangles). As a consequence, the Passive Boundary sensors are added to the data collection tree. The alert operation is represented by the dotted arrows.

from this elaboration, that are possibly filtered and reduced in size and amount, are finally send back to the sink, that will in turn project them to the user, as response of the query submitted.

These approaches exploit classical relational operators [41] and data base query processing techniques, appropriately adapted to the WSN data management context.

4.4. Differences of EQL with Tiny-DB like approaches

Although approaches such as TinyDB are not designed to track moving events, they can be adapted to this purpose. In particular it is possible to define a TinyDB query aimed at detecting an event in a given detection area. When the event is detected, the sink (and thus the user) starts receiving data related to the event from the sensors in the detection region. However, if the event moves outside the detection region, the sensors may lose the focus on the event. For this reason, the sink should track the movements of the event and should submit a new query issued to sensors located in the area where the event is moving.

It should be observed that this way of tracking events, which is feasible also with other approaches beyond TinyDB (for instance with MaD-WiSe), requires a centralized management of the tracking (executed by the sink or by the user). For this reason, we denote this approach with CQA (Centralized Query Approach).

On the other hand, the process of query migration to follow the movement of an event is primitive in EQL. This fact has a strong implication on the design of the query processor. Specifically, the activity of query migration is run by a sensor elected in the event area rather than in the sink, and in EQL are introduced the states of boundary active and boundary passive that are necessary to detect the movements of the event outside the event area to drive the migration of the query accordingly.

A second difference in the design of EQL is that it is possible to define chains of events, i.e. events that may cause other, different (but related) events. This feature allows for the definition of tracking tasks of event that are consequences of other previous events, and it is not present in TinyDB or in other proposals.

A further important difference with TinyDB-like approaches is in the level of abstraction of the query. In fact, a TinyDB query addresses sources of data that are individual sensors, while, as discussed in Section 3, an EQL query may address the events themselves as sources of data.

4.5. Execution Model

Figure 3 depicts a high-level abstraction of the Finite State Machine that represents the status of sensors while processing an EQL query. Each sensor passes through three states, *idle*, *detection* and *tracking*, which are actually super-states that are refined by several internal states. Note that a sensor can be involved in more than one query.

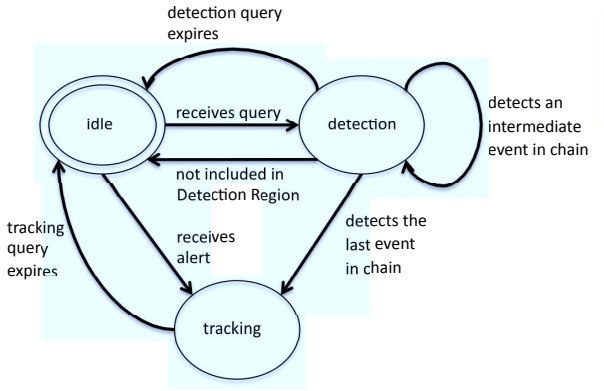


Figure 3: AUTOMATON - Representation of an abstraction of the Finite State Automaton of EQL processing. The *detection* and *tracking* states are complex super-states that are composed of several internal states.

However the FSM described in this section aims to show how the proposed solution works, thus it refers to the execution of just one query processing.

Initially each sensor is in the *idle* state, with its internal status set to passive (see Section 4.2 and Table 14). In this state, when a sensor receives a query, it passes to the *detection* state and checks if it is included in the Detection Region specified in the query. A sensor that belongs to the Detection Region sets its internal status to active (either inner or boundary); otherwise, it switches back to the *idle* state. In the *detection* state each sensor scans the Chain of Events and begins the detection of the next event in the chain. If a sensor detects the current event, it initiates the detection of the next event in the chain; otherwise it switches to the *idle* state and sets its current status to passive.

When a sensor detects the last event in the chain, it passes to the *tracking* state because it belongs to the Event Area of the event to be tracked (see Figure 4). In this state, it periodically acquires the data from the event to be sent to the leader. The leader, besides sampling the local readings, aggregates the received data and sends the result to the sink.

The *tracking* state can also be reached from the *idle* state as consequence of the reception of an alert message. This message is sent by the active boundary sensors, and only the passive sensors (that are in *idle* state) handle this message. In this case, they pass to the *tracking* state, become passive boundary and start executing the tracking task. If a passive boundary sensor in the *tracking* state detects the event, then becomes active and remains in this state, otherwise, after the timer of the tracking query expires, it switches back to *idle* state and becomes passive again.

As anticipated, detection and tracking states are super-states that contain more refined states. In the following, we describe these refined states:

Detection State Refinement

A first refinement of the *detection* super-state is presented in Figure 4. For simplicity, we limit the discussion to this level of refinement; further levels of refinements are straightforward.

The input event to this super-state is the reception of a query that is firstly handled in the *checking inclusion* state. In this state, the sensors check if they belong to the Detection Region defined in the query. Sensors not included in the specified Detection Region exit from this super-state and switch back to *idle*, the other sensors switch to the detection of the 1^{st} event in the Chain of Events. If the sensors in this state do not detect the event within a timeout, they go back to the *idle* state, otherwise they become active (either inner or boundary) and switch to the detection of the next event in the chain. When the sensors detect the last event in chain, they go to the *tracking* super-state.

Tracking State Refinement

The refinement for the *tracking* super-state is reported in Figure 5. In the first state of the tracking task (*electing leader*) the leader is elected. This state forks the execution of the task in two branches, one executed by the leader and the other one executed by all the other sensors. The leader first initiates the building of the data collection tree, and then it executes a loop of two states in which it samples the local transducers, aggregates data coming from the sensors in the tree, and sends the aggregated data to the sink. The leader exits from this states if the tracking timeout

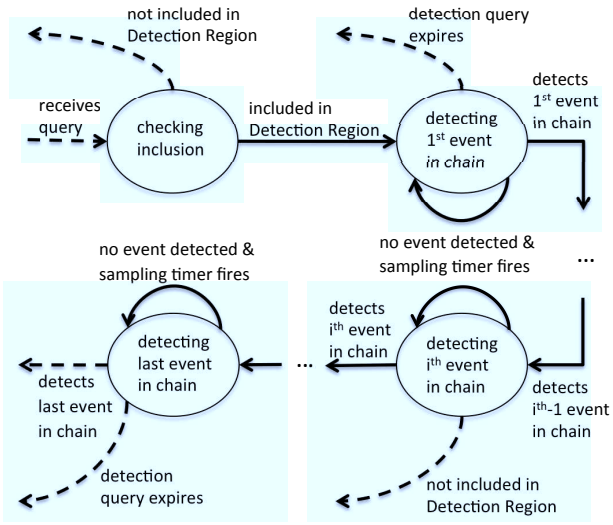


Figure 4: Detection State Refinement - Refinement to the *detection* super-state of the FSA reported in Figure 3. The dashed lines represent the input/output transitions to/from the *detection* super-state.

expires or if the event moves and it is not included in the Event Area anymore. For lack of space, we do not report the states related to the maintenance of the tree (including the election of a new leader).

The branch executed by the other sensors is similar to the branch executed by the leader. These sensors first participate to the tree-building phase. Data collection is obtained by executing a loop in which sensors acquire data from the event and send them to the leader. A sensor exits from this state if the tracking timer expires or if it does not detect the event anymore. In the former case, it switches to the *idle* state, whereas in the latter one it goes to the *checking inclusion* state. In this state, it continues detecting for the event until the tracking timeout expires. When this happens, it switches to *idle*.

Finally, a sensor reaches the *passive boundary* state from *idle* when it receives an alert message by an active neighbor sensor. In this state, the sensor checks for the conditions related to the event to be tracked. If the sensor detects the event, then it is included in the data collection tree and it switches to the data-sampling loop.

5. Evaluation

In this section, we present the evaluation of the proposed system in terms of power consumption and of the percentage of successfully tracked events. We also compare the power consumption of EQL with the power consumption of executing the detection and tracking of a moving event with a centralized query approach (CQA). In this latter approach, the sensors acquire the data and send them to the sink that performs the detection and pilots the tracking of the event. This approach is based on the TinyDB query processor adapted (with some abstraction) to the event detection scenario and to the use of a geographic routing protocol. In particular, in CQA we assume that the sink deduces the direction and the speed of the event, so that it submits a new query to the sensors displaced in the area that will be covered by the event every time it moves of a distance equals to its radius.

In order to compute the power consumption of the two systems, we present a cost model that takes into account the communications among sensors and the activations of the transducers, and we present the MATLAB simulator that we used to perform the experiments. By means of the simulator, we evaluate the movement of the events and the consequent migration of the queries over a reasonably large scale, with variable speed and size of the events, a thing that would not be easily achievable with limited-scale testbeds. Specifically, in the simulations we consider events with circular shape and variable sizes and with uniform rectilinear motion at different speeds. We evaluate the power consumption of EQL queries and CQA queries by using the cost model described in the next section and the implementation model given in Section 4.

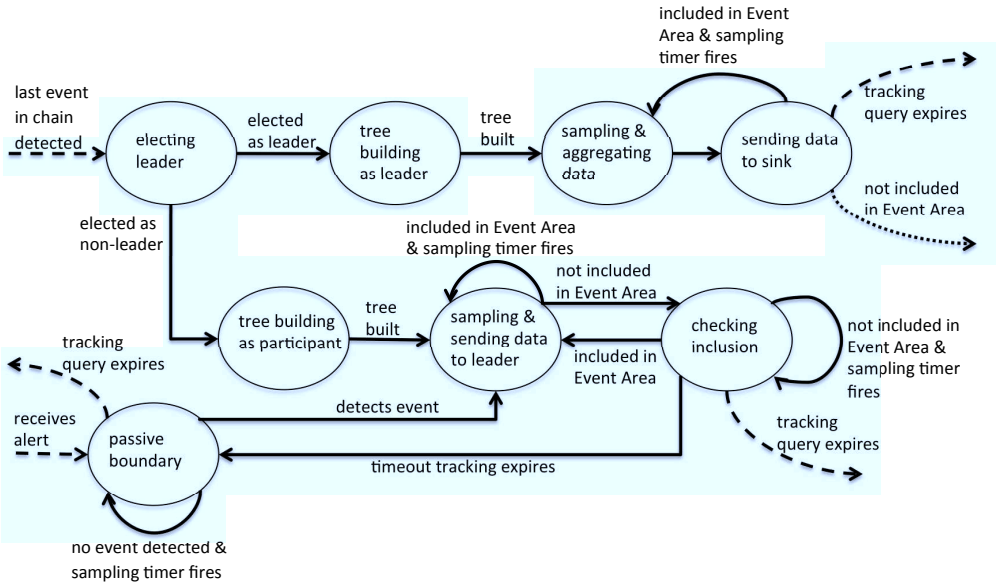


Figure 5: TRACKING STATE REFINEMENT - The dashed lines represent the input/output transitions to/from the *tracking* super-state. The dotted outgoing line should lead to the states related to the operations needed to update the leader and to maintain the data collection tree that are not reported for lack of space.

Transducer	Energy per Sample (mJ)
Accelerometer	0.03222
Magnetometer	0.2685
Light	0.00009
Temperature	0.00009

Table 15: Energy required for a sample from various transducers of MTS310CA boards.

5.1. Cost Model

The cost model takes into account the number of messages transmitted and received by the sensors and the number of the transducers activations during the execution of an EQL and a CQA query. Regarding the communications, we take into account only the overhead caused by the actual transmission and reception of a message of 50 bytes with respect to the power consumption of keeping the radio active. We disregard the cost of sending the tracking data to the sink and the cost of the internal computation of the sensors, since these costs are equivalent for both the systems and they are negligible compared to the other costs. For the sake of simplicity, we also assume that all messages have the same size.

We assume that IRIS mote with MTS310CA sensor board for the transducers [42] are used to implement the query and we evaluate the costs referred to this hardware platform. Tables 15 and 16 report the related costs.

5.2. Experimental Environment

We consider a network composed of sensors uniformly distributed over a squared area of size $L \times L$ meters (see Figure 6), where each sensor has a fixed transmission range r_x, an average number of neighbors n_x, and it is aware of its position. We assume that the sink is placed in the center of the network and that a geographic routing protocol is used to route packets. By simple geometric arguments, we can deduce the total number of sensors in the network (N_{tot}):

$$N_{tot} = \frac{n_x}{(\pi r_x^2)} L^2$$

Operation	Energy required (mJ)
c(Send)	0.1494225
c(Rec)	0.16917375

Table 16: Energy required for sending and receiving a message of 50 bytes on the IRIS mote.

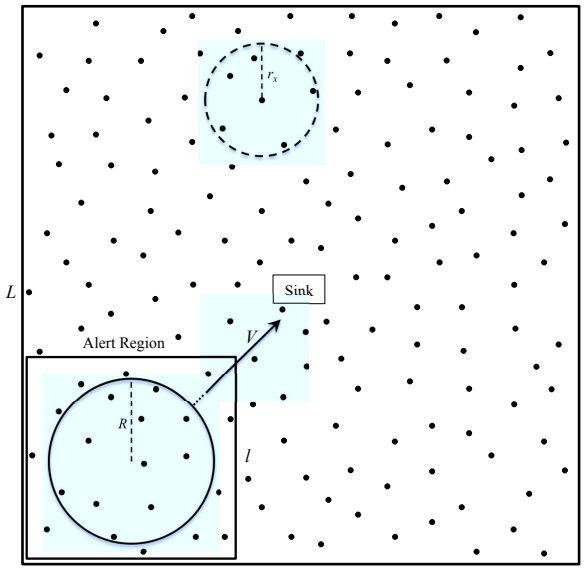


Figure 6: NETWORK - Each sensor has a transmission range r_x. The small square represents the Detection Region. The circle represents the Event Area and it moves according to the rectilinear motion V. The sink is assumed to be in the center of the network.

We assume that the Detection Region is a squared area with side l placed in the bottom-left corner of the network and that the Smallest Event Size is of one hop.

We model an event as a circle with radius R that moves with uniform rectilinear motion V. Without loss of generality, we assume that the event moves from the Detection Region to the opposite corner of the network.

The simulator has the following parameters: number of neighbors per node, size and speed of the event, sampling rate of the sensors and timeout duration. We make simulations in different scenarios: in each scenario, we vary one parameter of the simulator while keeping fixed the other parameters. For each scenario, we perform 50 independent experiments on different randomly generated network distributions.

Table 17 reports the list of parameters that we kept fixed in our simulations, and Table 18 shows the parameters that vary during the simulations.

5.3. Cost of Event Query Language

In this section, we evaluate the cost, in terms of energy consumption, of the detection of an event $ev1$ and of the tracking of an event $ev2$ dependent on a ev1, with EQL. To this purpose, we analyze all phases executed during the detection and tracking (we assume that the query has already been injected), which are: Detection, Alert, Check Event, and Tree Updating. Note that the last three phases are executed periodically, according to the sampling period Δt.

Detection (detection of the next event in the Chain of Events) - Once the query is received, the sensors in the Detection Region start monitoring the conditions that describe the first event in the Chain of Events. In particular, each sensor acquires the needed data from the local transducers and sends them in a local broadcast to its 1-hop neighbors (recall that we assume that the Smallest Event Size is of one hop). Therefore, each sensor computes the average value of the received data and determines if the event occurred. Let A_r be the set of sensors in the Detection Region, the cost of one iteration of the detection c_{dt}^{EQL} is given by the cost of transducers activation times the sensors in A_r plus

Name	Description	Value
L	length of the network	1000 m
l	length of the Detection Region	20 m
r_x	sensor's transmission range	10 m
γ	transducer activation cost	0.3 mJ
β	unicast communication cost	$c(Send) + c(Rec)$
δ	broadcast communication cost	$c(Send) + n_x c(Rec)$
T	life time of the event	1800 sec

Table 17: Fixed parameters used for the analysis.

Name	Description
n_x	number of neighbors per sensor
R	Event Area radius
v	event speed
exp	timeout for detecting the event
Δt	sampling rate

Table 18: Variable parameters used for the analysis. In each scenario, we study the behavior when changing one of this parameters and keeping the other ones to a fixed value.

the cost of a broadcast communication times the sensors in A_r, i.e., $c_{dt}^{EQL} = (\gamma + \delta)|A_r|$, where $|A_r|$ is the cardinality of A_r. This is repeated, according to the specified sampling rate, until the event occurs.

Alert (alerting of the neighbor sensors to be ready to detect the event) - The alert is used to involve in the tracking task the sensors around the Event Area that are not already involved in this task. Specifically, all the sensors around the Event Area are alerted, in an area as large as specified in the `EVOLUTION` clause of the Tracking Statement.

Except for the first iteration (where all the active boundary sensors execute the alert operator, see Figure 7), at each iteration only the new active sensors (i.e. the sensors that entered in the Event Area at the previous iteration) alert their neighbors (see Figure 8). We call these sensors *Added Sensors*, and we denote this set by D_s. Each of these sensors sends a broadcast alert message up to the distance specified in the Tracking Statement. Therefore the cost of the Alert phase c_{al}^{EQL} is given by the cost of a broadcast communication times the sensors in D_s, i.e., $c_{al}^{EQL} = \delta|D_s|$.

Check Event (detection, by the alerted sensors, of the currently tracked event in order to follow its evolution) - During the tracking task the active and the alerted sensors periodically monitor the conditions that define the event either to check if they are still involved in it (for the active sensors) or to check if they have just been included in the area covered by the event (the alerted sensors). Let C_s and L_s respectively be the set of active and alerted sensors, the cost of the Check Event phase c_{ck}^{EQL} is given by the cost of transducers activation times the number of sensors in the two sets C_s and L_s, i.e., $c_{ck}^{EQL} = \gamma|C_s \cup L_s|$

Since the data to be returned to the user (i.e. the data specified by the `SELECT` clause of the Query Statement) are a subset of the data needed to check the event, the data collection task is implicitly executed during this phase.

Tree Updating (updating of the Event Area and the data collection tree) - Due to the event movements the sensors that can detect the event are only a subset of all the alerted sensors (i.e. D_s). In particular, only the sensors displaced in the lune-shaped region between the Event Area at time t and the Event Area at the time $t + \Delta T$ will be covered (see Figure 9). These sensors thus become active and they are added to the data collection tree to start executing the tracking of the event. To this purpose, they send a unicast message to one of their active neighbor sensors as acknowledgement of the actual event detection. Thus, the cost of the Tree Updating phase c_{tu}^{EQL} is given by the cost of a unicast communication times the number of sensors in D_s, i.e., $c_{tu}^{EQL} = \beta|D_s|$.

5.4. Cost of CQA

As said in in Section 5, we compare the power consumption of EQL with the power consumption of a centralized query approach (CQA) that is based on the TinyDB query processor. In this approach, the sensors acquire the data from the event and send them to the sink that performs the actual detection and pilots the tracking of the event. The

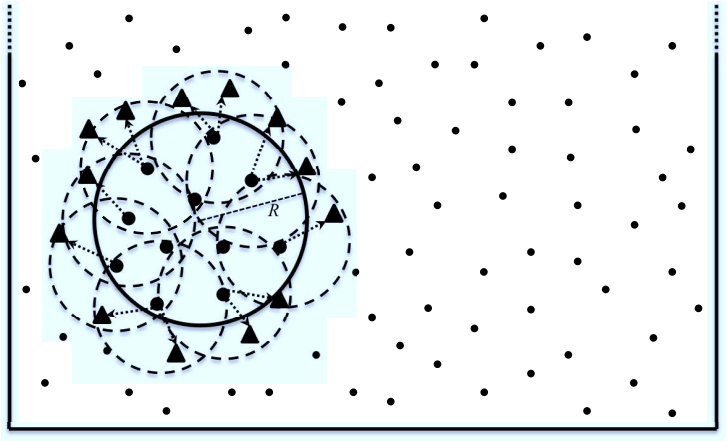


Figure 7: First Alert - The solid circle is the Event Area with radius R. The dashed circles represent the transmission of the alert messages and the triangles are the alerted sensors.

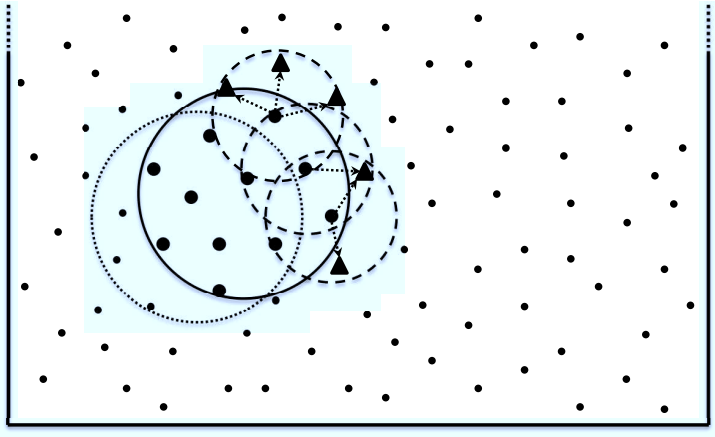


Figure 8: Alert - The added sensors send the alert message.

detection task executed in CQA is similar to that executed in EQL, while the tracking task of CQA is different from that of EQL since in CQA the sensors are not able to autonomously follow the event, but new queries have to be submitted according to the movements of the event. In particular, the phases executed by CQA are: Detection, Query Submission, Query Broadcast, and Data Acquisition.

Detection (detection of the next event in the Chain of Events) - This phase is executed in the same manner as EQL and has the same cost.

Query Submission (submission of the query to the center of the new Event Area) - In CQA the sink has to renew the query when the event has moved out the region covered by the previous query. To do this effectively, the sink needs to know the speed at which the event moves, and, without this information, it may lose the event or it may be forced to inject more queries than needed. To the purpose of comparison against EQL, we assume that the CQA sink has the information about the event speed, so that it can inject a new query as soon as the event has moved at a distance twice the radius of the event ($s = 2r$). Note that this is the minimum rate at which the sink can submit a new query, not missing the event. The message containing the query is sent to the sensors in the Event Area by means of unicast transmissions using the path used to collect the data from the event. In order to estimate the cost of this operation, we compute the distance d_x between the sink and the center of the event. With this information, and by knowing the transmission range r_x of the sensors, we determine the number of hops needed to forward the query on

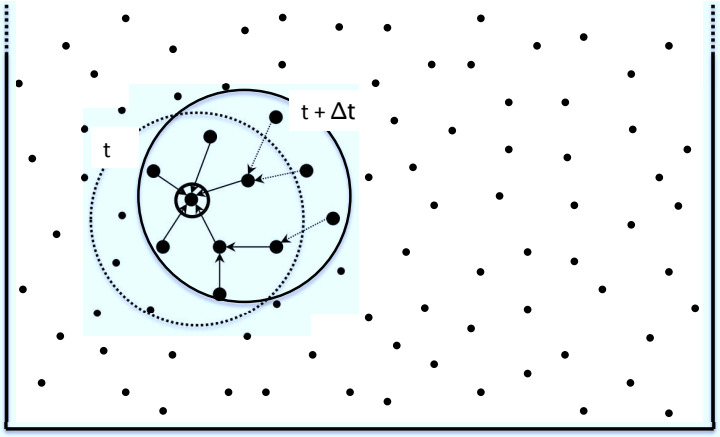


Figure 9: TREE UPDATING - The solid circle represents the Event Area at time $t + \Delta t$, while the dotted circle represents the Event Area at time t. The newly involved sensors in the lune are added to the data collection tree (the dotted arrows).

a path of length d_x. Therefore, the cost of the query submission in CQA c_{qs}^{TDB} is given by the number of the sensors thus computed times the cost of a unicast communication, i.e., $c_{qs}^{TDB} = \beta\lceil d_x/r_x \rceil$

Query Broadcast (broadcast of the query inside the Event Area and building of data collection tree) - Once the query arrives to the center of the Event Area, a local broadcast is executed inside this area to forward the query to all the active sensors and also to alert the sensors around the event. This is needed to let the sink understand in which direction the event is directed and to submit the query at the following iteration. We call F_s the set of the sensors involved in this broadcast communication. This phase is also used to build an updated data collection tree. This phase has a cost c_{qb}^{TDB} equal to the cost of a broadcast communication times the number of sensors in F_s, i.e., $c_{qb}^{TDB} = \delta|F_s|$

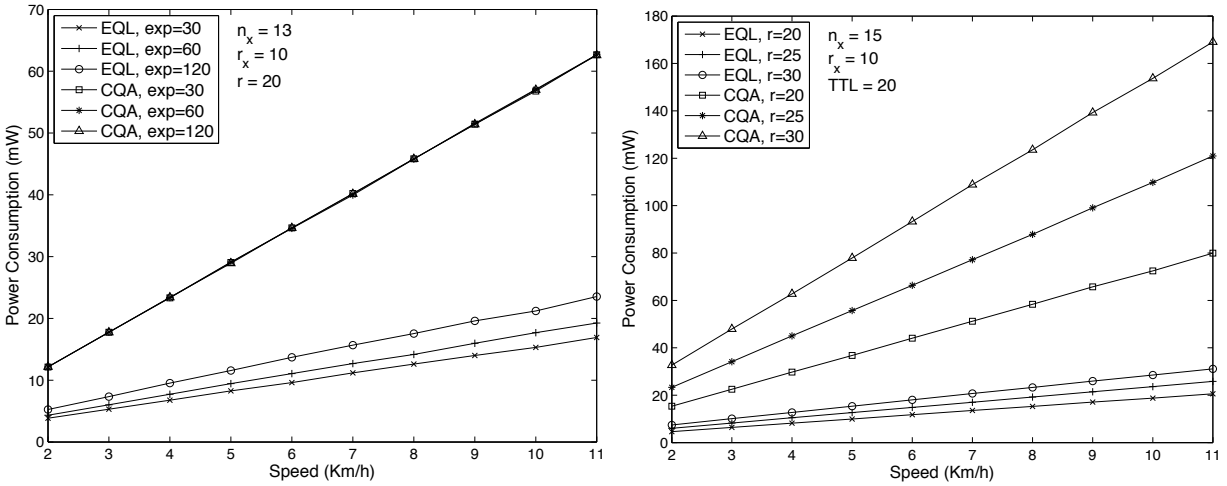
Data Acquisition (acquisition of the requested data) - When the Query Broadcast phase ends, the sensors in F_s activate the transducers and acquire the requested data from the event. The cost of this phase c_{da}^{TDB} is given by the cost of the transducers activation times the number of sensors in F_s, i.e., $c_{da}^{TDB} = \gamma|F_s|$. This phase is executed periodically (according to Δt parameter) for the whole lifetime of the query, that is the time needed by the event to move of a distance $s = 2r$.

5.5. Results

In order to evaluate the power consumption (in terms of energy consumed by all sensors in a unit of time) of our system compared to CQA, we analyze different scenarios, by combining the parameters reported in Table 18. All the results presented in this section are obtained with a 95% confidence interval. For each scenario, we performed 50 iterations, each time generating a new network topology according to the uniform distribution of the density of the sensors. Therefore, each point plotted is the average value of power consumption computed over the 50 iterations. In all the experiments we vary the value of the analyzed parameters, while we fix the value of the other parameters as follows: number of neighbors per node $n_x = 13$, radius of the event $r = 20$, event speed $v = 2Km/h$, sampling rate of the sensors $\Delta t = 2seconds$, transmission range of the sensors $r_x = 10mt$ and `TIMEOUT` value $exp = 20seconds$.

In Figure 10, we report the power consumption of the two systems by combining increasing values of event speed and different values of expiration time and size of the Event Area. The graphs show that for both systems the power consumption is higher as the event speed increases since more new sensors, in each sampling cycle, are involved in the event. This implies more communications and more transducer activations. In both experiments, EQL shows a lower power consumption than CQA and it scales better as the studied parameters increase. This because, when a new query is submitted in CQA, more sensors are involved in the query thus implying a higher power consumption. On the other hand, in EQL all the tracking operations are localized to the nodes in the boundary of the Event Area, and this limits the overall power consumption with respect to CQA.

In Figure 10(a), we note that CQA is independent of the expiration time since in this case the sensors are not alerted, while in EQL a higher value of this parameter produces a higher power consumption since the sensors remain



(a) Power consumption in the speed-expiration time scenario.

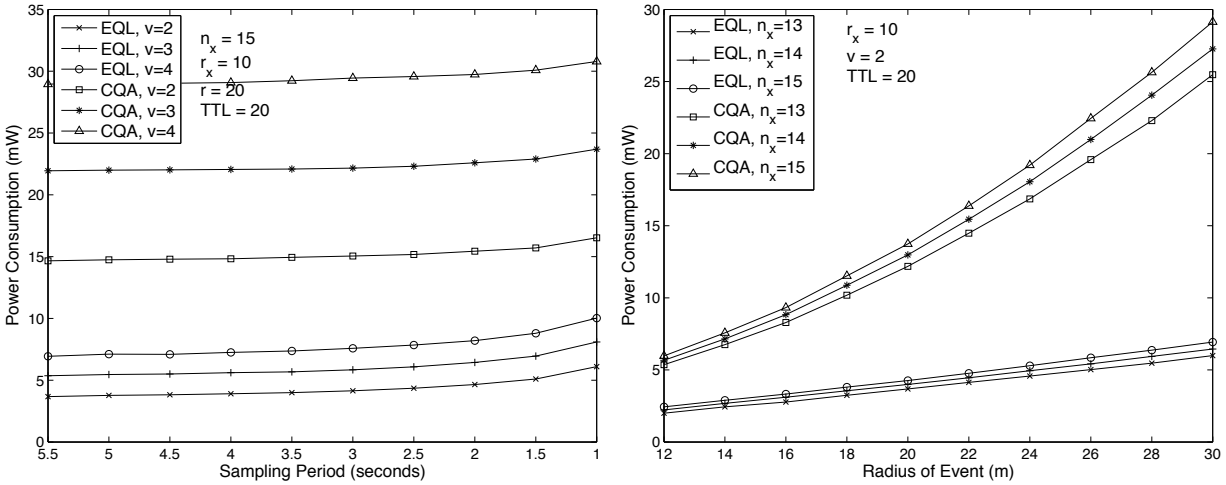
(b) Power consumption in the speed-size of event scenario.

Figure 10: Power consumption with increasing value of event speed and different values of expiration time (a) and size of the Event Area (b).

active in the monitoring phase for a longer time. In any case, with an expiration time that ranges between 30 and 120 seconds, EQL has a power consumption that ranges from about 4 mW to about 20 mW. On the other hand, although CQA is independent of the expiration time, its power consumption is much higher and it increases linearly with the speed.

In Figure 10(b), we can see the power consumption of EQL and CQA in the scenario where the speed and the size of the event are studied. In this case, a bigger event implies a larger number of sensors involved in it, and this produces a greater power consumption. CQA shows a higher power consumption with respect to EQL, which also grows more than EQL as the radius of the event increases, since submitting a query in a larger area is more expensive. EQL, on the other hand, is slightly affected by the size of the event, since the operations executed to track the event are local to the boundary nodes of the event, and a wider event does not imply too much overhead in this case. For example, with an event of radius 20 meters, the power consumption of EQL ranges from 4.6 mW, if the event speed is 2 Km/h, to 20.6 mW, if the event speed is 11 Km/h, while the power consumption of CQA, with the same size and same event speed, ranges from 15.3 mW to 80 mW.

Figure 11(a) reports the power consumption of EQL and CQA in the scenario where we increase the sampling rate of the sensors with different values of event speed. We can see that for both systems the power consumption is higher as the sampling rate increases (i.e. the period between two consecutive sampling decreases) because the transducers are activated more frequently. The power consumption also grows as the event speed increases. This happens because, as noticed before, if the event is quicker more new sensors become involved in the event between two consecutive sampling, and thus more communications have to be executed. We can notice, however, that the impact of the event speed is bigger than the impact of a higher sampling rate, because the communications caused by a quicker event are more expensive than the activations of the transducer. The figure also shows that the impact on the power consumption when increasing the sampling rate of the sensors is smaller in CQA than in EQL, because in CQA only the sensors in the Event Area activate their transducers, while in EQL also the alerted sensors execute the sampling, thus consuming more energy. In fact, for an event speed of 2 Km/h, the power consumption of EQL ranges from 3.7 mW, with a sampling rate of 5.5 seconds, to 6.1 mW, with a sampling rate of 1 second, while the power consumption of CQA ranges from, 14.7 mW to 16.5, with the same sampling rate and event speed. That means a growth of the power consumption of 0.54 mW/s in case of EQL, against a growth of 0.41 mW/s in case of CQA. With an event speed of 3 Km/h, EQL has a power consumption of 5.4 mW, if the sampling rate is 5.5 seconds, and it is 8.1 mW, if the sampling rate is 1 second, while, with the same sampling rate and the same event speed, CQA has a power consumption that ranges from 21.9 mW to 23.7 mW. In this case, the growth of the power consumption in EQL is 0.61 mW/s, while in CQA it is 0.39 mW/s. With an event speed of 4 Km/h, the power consumption of EQL ranges from 6.9 mW, with



(a) Power consumption in the sampling rate-event speed scenario. (b) Power consumption in the size of event-network density scenario.

Figure 11: Power consumption with increasing value of sampling rate of the sensors and different values of event speed (a) and increasing size of the Event Area and different values of network density (b).

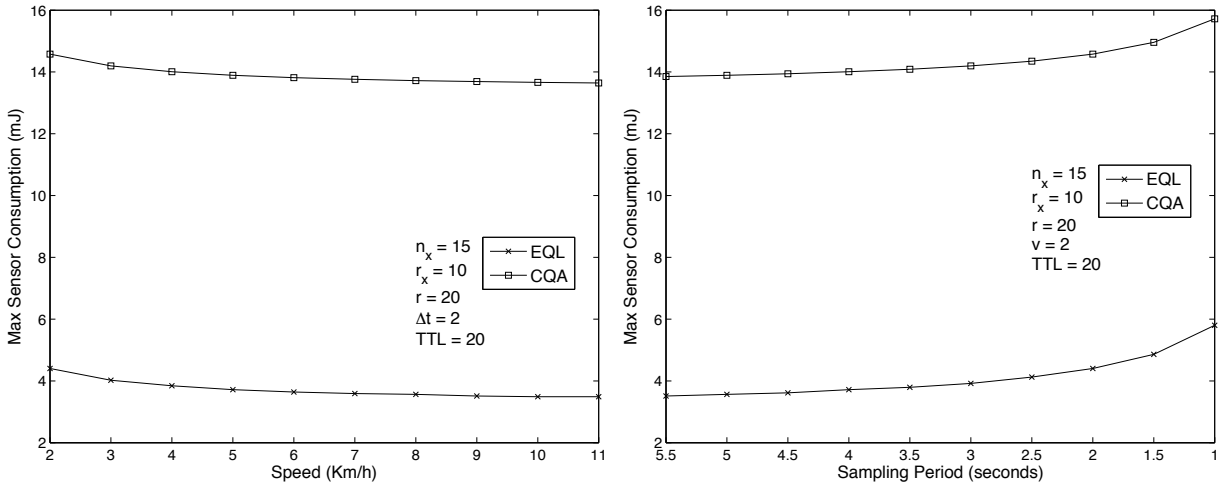
a sampling rate of 5.5 seconds, to 10 mW, with a sampling rate of 1 second, while the power consumption of CQA, with same sampling rate and event speed, ranges from 28.9 mW to 30.8 mW. In this case, the growth of the power consumption in EQL is 0.69 mW/s, while in CQA it is 0.41 mW/s.

In Figure 11(b), we analyze the power consumption of EQL and CQA for increasing values of the size of the Event Area and different values of network density. We can see that the power consumption increases with the size of the event, but this affects much more CQA than EQL, since a wider event involves more sensors that have to receive the query, so more communications are needed. While in EQL the tracking of the event is executed locally by the sensors at the border of the event, with a much smaller overhead required. The figure also shows that the network density affects both systems. However, this has a bigger impact on CQA than on EQL, since, also in this case, more sensors are involved in the event. For example, with an average number of neighbors per node equal to 13, the power consumption of EQL ranges from 2 mW, with a radius of the event of 12 mt, to 6 mW, with a radius of the event of 30 mt. While the power consumption of CQA, with the same network density and same size of the event, ranges from 5.4 mW to 25.5 mW.

We also analyze the energy consumption of the individual sensors and we report, in Figure 12, the total energy consumption of the sensor that consumes most energy in the scenarios where we increase respectively the event speed and the sampling rate of the sensors.

Figure 12(a) reports the maximum energy consumption of a sensor as the event speed increases. Despite what happens with the power consumption, that is independent of the time and thus it increases with the event speed (see Figure 10), the energy consumption decreases as the event moves quicker, because each sensor is involved in the event for less time, thus wasting less energy. The graph shows also that a sensor in EQL has smaller energy consumption than one in CQA because a sensor in CQA is, in general, involved in more communications than a sensor in EQL. However, we can notice also that the energy consumption decreases more deeply in EQL than in CQA as the event speed increases. In fact, the energy consumption in EQL ranges from 4.4 mJ, with an event speed of 2 Km/h, to 3.5 mJ, with an event speed of 11 Km/h, that is a decreasing of 0.102 mJ/Km/h. The energy consumption in CQA, instead, with the same values of the event speed, ranges from 14.6 mJ to 13.6 mJ, that is a decreasing of 0.104 mJ/Km/h.

In Figure 12(b), we report the maximum energy consumption of a sensor with increasing value of the sampling rate of the sensors. In addition, in this case EQL shows a smaller energy consumption, with respect to CQA, because of the less communications. However, in this case, the energy consumption of EQL grows faster than that of CQA. This happens because in CQA only the sensors in the Event Area activate their transducers, since there is no alert of the sensors, while in EQL also the alerted sensors perform the sampling, thus wasting more energy as the sampling rate increases. In fact, the energy consumption of EQL ranges from 3.5 mJ, with a sampling rate of 5.5 seconds, to



(a) Maximum sensor consumption in the speed scenario.

(b) Maximum sensor consumption in the sampling rate scenario.

Figure 12: Maximum energy consumption of the sensors with increasing values of expiration time (a) and sampling rate of the sensors (b).

5.8 mJ, with a sampling rate of 1 second, that is an increment of 0.416 mJ/s. CQA, on the other hand, has an energy consumption that ranges, for the same values of sampling rate, from 13.8 mJ to 15.7 mJ.

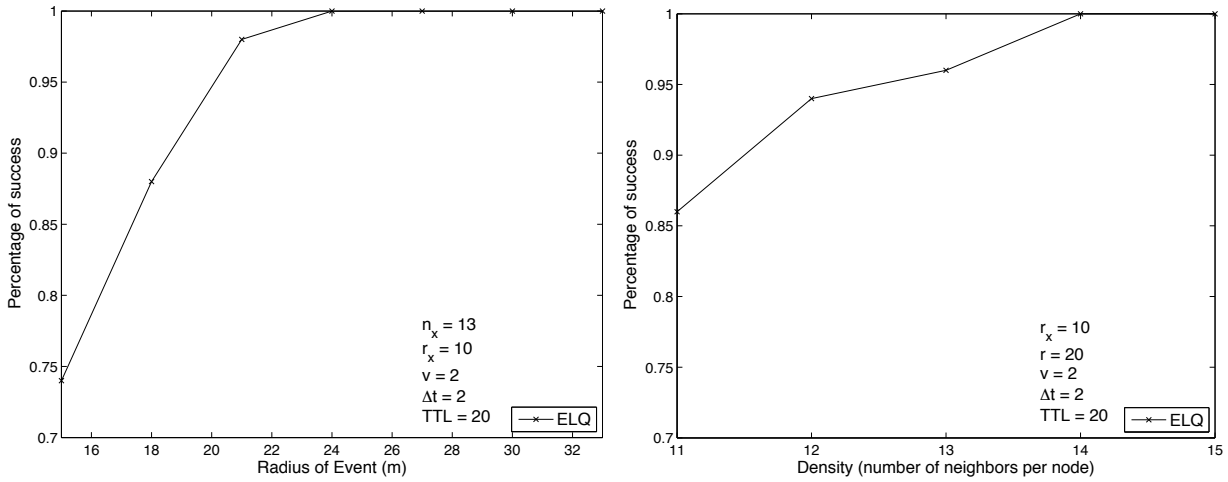
We now evaluate the percentage of successfully tracked events. The purpose of this analysis is to find a proper calibration of the parameters in order to ensure that the events are tracked successfully with a low energy consumption.

The event can be lost because of the following reasons: (i) the event is too slow and the expiration time is not large enough, so the alerted sensors stop monitoring for the event before it reaches them; (ii) the event is too fast and the sampling rate too short, so the sensors do not detect it between two consecutive samplings; or (iii) the event reaches a hole in the network where no sensor can detect it.

We first analyze the percentage of successfully tracked events for some parameters that vary individually, then we analyze combinations of varying parameters. Figure 13 shows, the percentage of success with respect to the size of the Event Area and the network density, respectively. In these cases, the event may be lost when the event traverses areas of the network in which there are no sensors (network holes). With a bigger Event Area, or a denser network the probability to lose the event for this reason becomes lower and lower. In fact, we can see in Figure 13(a) that with an event of radius 15 meters the percentage of success is of about 74% over 50 iterations, while the event is not lost at all when the radius of the event becomes bigger than 24 meters. Similarly, Figure 13(b) shows that the percentage of success with a number of neighbors per node equals to 11 is 86%, while it becomes 100% with a number of neighbors per node equals or greater than 14.

In Figure 14(a) the percentage of success is shown for different values of expiration time. In this case, if the expiration time is too short, the event is lost because the alerted sensors stop monitoring for the event before it reaches them. We can see in the figure that with an expiration time of 5 seconds the event is always lost, but the percentage of success increases as the expiration time grows and the event is never lost with an expiration time equals or greater to 15 seconds. Of course, the expiration time is strictly related to the event speed. If the event moves with a high speed, even a low value of the expiration time can be sufficient to successfully track the event. In Figure 14(b) we analyze the percentage of success with a combination of these two parameters. With an expiration time of 15 seconds, an event speed of 2 Km/h is enough to guarantee a success rate of almost 100%. With an expiration time of 10 seconds, the event should move at least 3 Km/h in order to be successfully tracked with a probability of more than 90%, while with an expiration time of 15 seconds the speed grows to 6 Km/h in order to have a success rate greater than 90%. Therefore, the expiration time should be set accordingly to the event speed, if this information is available.

The expiration time also affects the power consumption of EQL, since a high value of this parameter means that the alerted sensors that are not involved by the event, remain monitoring for it for a longer time, thus wasting more energy. Figure 15 shows that a value of the expiration time greater than 700 leads to a power consumption higher



(a) Percentage of success in the size of event scenario.

(b) Percentage of success in the density scenario.

Figure 13: Percentage of successful tracking with increasing value of the size of the Event Area (a) and network density (b).

than that of CQA (we recall that CQA is independent of the expiration time, because in that case the sensors are not alerted, therefore the line related to CQA is flat). A larger value of this parameter ensures a higher success rate, but it also implies a greater power consumption of the sensors. There is a trade-off between the percentage of successful tracking and the power consumption of EQL related to the expiration time.

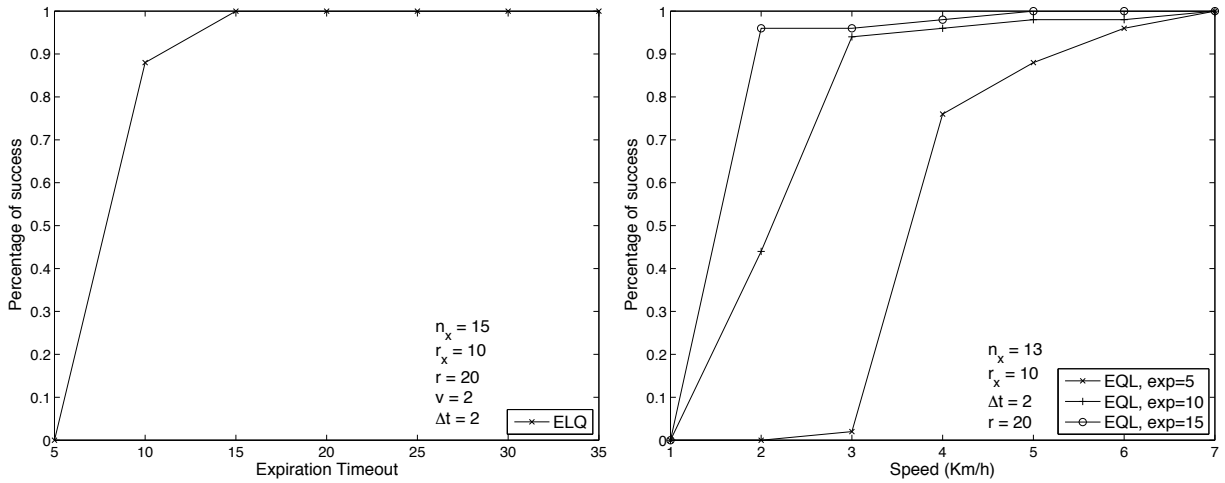
Figure 16(a) shows the relation between the event speed and the sampling rate of the sensors. If the event is too fast and the sampling rate is too short, it can happen that the alerted sensors monitor for the event before it reaches them, and then they perform the subsequent sampling when the event has already passed. Also in this case the event is lost. We can see in the figure that for a sampling rate of 2 seconds the success rate of the tracking is 100% if the event speed is equal or less than 4 Km/h, while it is always lost if the event moves at a speed equal or greater than 8 Km/h. With a sampling rate of 4 seconds the event is successfully tracked with a probability of 100% if it has a speed of 8 Km/h or less, while it is always lost if it moves at 10 Km/h or more. With a sampling rate of 6 seconds, the event can move at a speed of at most 14 Km/h and it is still successfully tracked with a probability of 100%, while it is always lost if it moves at 18 Km/h or more. As the expiration time, also the sampling rate should be set accordingly to the event speed, if available (and of course to the application requirements).

Finally, Figure 16(b) analyzes the scenario in which the event is lost because it moves to a region of the network with no sensors. This can be avoided if, either the network is more dense, or the event covers a wider area. The figure shows that with the number of neighbors per node of 13, if the event has radius 12 meters it is successfully tracked only the 40% of times, while this reaches the 100% if the event has a radius of at least 22 meters. With the number of neighbors per node of 14, the successful tracking percentage grows to 74% with an event of radius 12 meters, and it reaches 100% if the event has radius 20. The 100% of successful tracking is reached with an event of radius 20 also in case of number of neighbors per node of 15, but in this case the probability of not losing the event with a radius equal to 12 is almost 90%.

5.6. Multiple Tracking and Events Chain

In this section we perform some experiments aimed at studying how the proposed system behaves when tracking multiple contemporary events and when dealing with chain of events of different size.

Regarding the multiple tracking, we tried to figure out how many events the system is able to track simultaneously on the basis of the occupation of communication channel within a sampling period. Figure 17(a) shows the result of this experiment. Given a certain radius of the event, we adjust the density of the network and the sampling period to guarantee that the system is able to handle 2, 4, and 6 spatially overlapping events. Therefore, this experiment represents a worst-case analysis. In fact, since the tracking in EQL is totally distributed, we are able to cope with an indefinite number of simultaneous events that do not spatially overlap. As it is possible to see from the graph, for the



(a) Percentage of success in the expiration time scenario.

(b) Percentage of success in the speed-expiration time scenario.

Figure 14: Percentage of successful tracking with increasing value of the expiration time (a) and with a combination of speed and expiration time values (b).

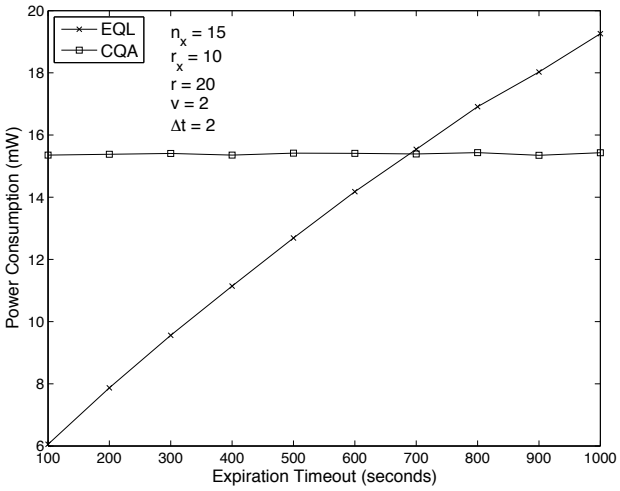
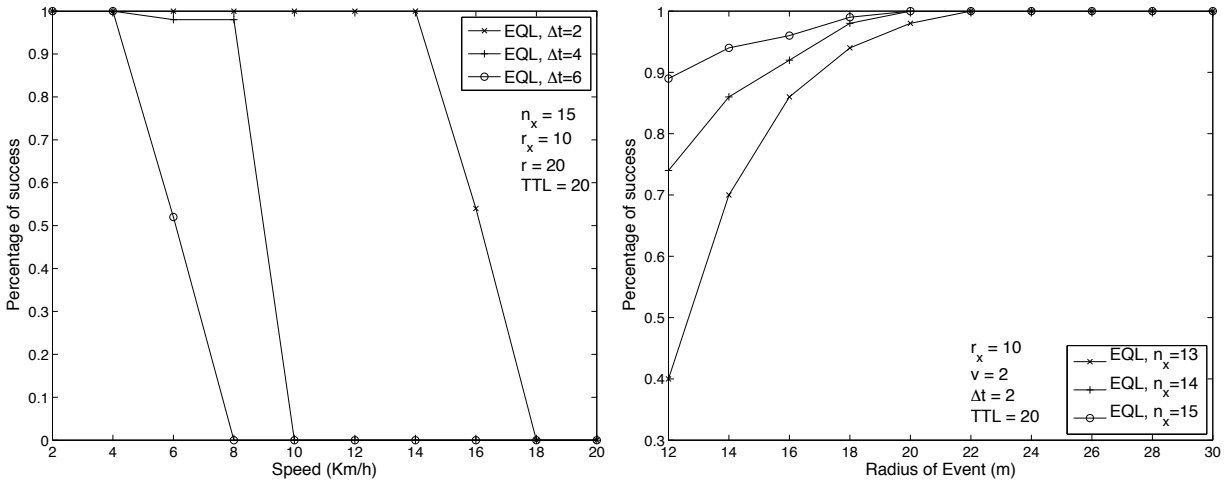


Figure 15: Power consumption with increasing value of the expiration time.

three configurations of the event radius and density of the network, in order to increase the number of overlapping events we need to increment the sampling period. This increment is more pronounced when the radius of the event gets larger (although we have reduced the density of the network); this is because more nodes are involved and so there are more communications

Chain of Events is a mechanism that allows developers to define an event that depends on a previously defined event. This mechanism does not concern the event tracking but only the initial event detection. When an event *A* that has been defined as trigger for second event *B* occurs the system starts waiting for the second event *B* until a user specified timeout expires. If the sensors in this state do not detect the event *B* within a timeout, they go back to the idle state, otherwise they become active (either inner or boundary) and switch to the detection of the next event in the chain. We have studied the impact of the timeout in a scenario where a chain of events occurs randomly with exponentially distributed arrival time and fixed average inter-arrival time. Figure 17(b) shows the "timeout multiplication factor", which represent how many times the timeout must be larger than the average inter-arrival time in order to have to ensure an average of 90% success in following the complete chain of events. The graph plots



(a) Success rate in the speed-sampling rate scenario.

(b) Success rate in the size of event-density scenario.

Figure 16: Percentage of successful tracking with increasing values of the event speed and different values of sampling rate of the sensors (a) and with increasing size of Event Area and different values of network density (b).

the timeout multiplication factor for an increasing number of events. This experiment suggests how to increase the waiting time parameter to ensure that no event in the chain is missed. Naturally, this problem, once again, requires us to face a trade-off between energy consumption and reliability of the system.

6. Conclusion

Differently than systems of query processing in WSN (like TinyDB or MaD-WiSe), the Event Query language (EQL) is tailored to detection and tracking tasks of composite events. It allows the users to specify queries that include the description of the events, the logical relationship between consecutive events (Chain of Events), and the parameters related to the tracked event that need to be collected. The query processor of EQL is then capable of automatically migrating the query in the WSN in order to follow the event during its evolution. On the contrary, in other systems the user has to interact with the network by submitting a new query or modifying an already submitted query every time a new event occurs or a monitored one moves. Our analysis shows that EQL has also a lesser power consumption of the other query-based approaches in tracking moving events. As discussed in Section 5.5, EQL scales better than CQA (that is, power consumption increase is slower), as the number of sensors in the network and/or the speed of the monitored event increases. Our plan for future work is to extend this approach to more structured architectures (e.g. the internet of things) where heterogeneous connections and devices come into the picture.

References

[1] C. Intanagonwiwat, R. Govindan, D. Estrin, Directed diffusion: a scalable and robust communication paradigm for sensor networks, in: 6th annual ACM/IEEE international conference on mobile computing and networking, Boston, MA, USA, 2000, pp. 56–67.

[2] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, Tinydb: an acquisitional query processing system for sensor networks., ACM Trans. Database Syst. 30 (1) (2005) 122–173.

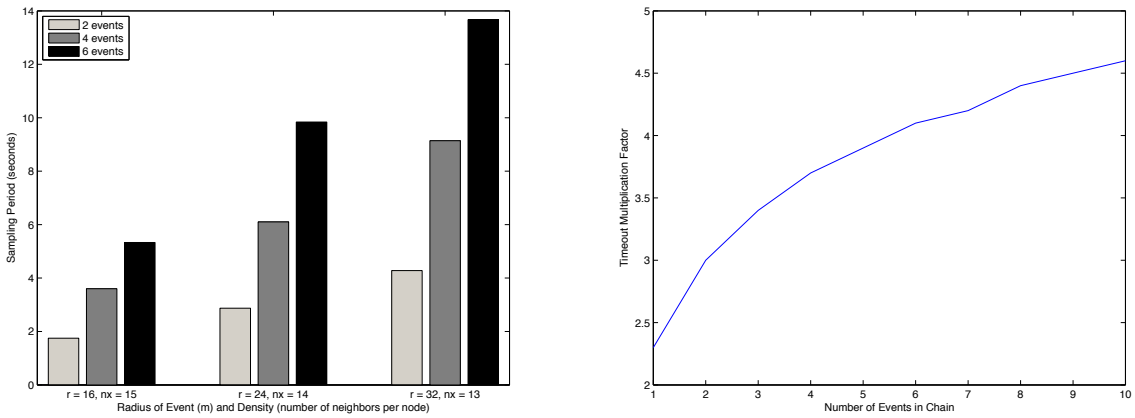
[3] Y. Yao, J. Gehrke, The cougar approach to in-network query processing in sensor networks., SIGMOD Record 31 (3) (2002) 9–18.

[4] G. Amato, S. Chessa, C. Vairo, MaD-WiSe: A distributed stream management system for wireless sensor networks, Software Practice & Experience 40 (5) (2010) 431–451.

[5] P. Baronti, P. Pillai, V. Chook, S. Chessa, A. Gotta, Y. F. Hu, Wireless Sensor Networks: a Survey on the State of the Art and the 802.15.4 and ZigBee Standards, Computer Communications 30 (2007) 1655–1695.

[6] J. Gehrke, S. Madden, Query processing in sensor networks, IEEE Pervasive Computing 3 (1) (2004) 46–55. doi:10.1109/MPRV.2004.1269131.
URL `http://dx.doi.org/10.1109/MPRV.2004.1269131`

[7] C. Vairo, G. Amato, S. Chessa, P. Valleri, Modeling detection and tracking of complex events in wireless sensor networks, in: IEEE Int. Conf. on Systems, Man, and Cybernetics (SMC), Istanbul, Turkey, 2010, pp. 235–242.



(a) Tracking of multiple Events.

(b) Event chain analisys.

Figure 17: Minimum sampling period needed to track multiple concurrent events with different size of Event Area network density (a), and timeout multiplication factor needed to successfully track event chains of different size.

[8] G. Amato, S. Chessa, C. Gennaro, C. Vairo, Dynamic tracking of composite events in wireless sensor networks, in: Ad Hoc Networks, Vol. 111 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer Berlin Heidelberg, 2013, pp. 72–86.

[9] A. V. U. Phani Kumar, A. M. Reddy V, D. Janakiram, Distributed collaboration for event detection in wireless sensor networks, in: 3rd Int. W. on Middleware for pervasive and ad-hoc computing, MPAC, 2005, pp. 1–8.

[10] D. J. Abadi, S. Madden, W. Lindner, Reed: Robust, efficient filtering and event detection in sensor networks, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, VLDB Endowment, 2005, pp. 769–780.
URL `http://dl.acm.org/citation.cfm?id=1083592.1083681`

[11] C. T. Vu, R. A. Beyah, Y. Li, Composite event detection in wireless sensor networks, 21st IEEE Int. Performance, Computing, and Communications Conference. 0 (2007) 264–271.

[12] M. Zoumboulakis, G. Roussos, Escalation: complex event detection in wireless sensor networks, in: Proceedings of the 2nd European conference on Smart sensing and context, EuroSSC'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 270–285.

[13] S. Kumar, K. K. R. Kambhatla, B. Zan, F. Hu, Y. Xiao, An energy-aware and intelligent cluster-based event detection scheme in wireless sensor networks, International Journal of Sensor Networks 3 (2) (2008) 123–133.

[14] S. Zhou, M.-Y. Wu, W. Shu, Improving mobile target detection on randomly deployed sensor networks, International Journal of Sensor Networks 6 (2) (2009) 115–128.

[15] Y. Yang, A. Ambrose, M. Cardei, Coverage for composite event detection in wireless sensor networks, Wireless Communications and Mobile Computing 11 (8) (2011) 1168–1181.

[16] M. Zoumboulakis, G. Roussos, Complex event detection in extremely resource-constrained wireless sensor networks, Mobile Networks and Applications 16 (2) (2011) 194–213. doi:10.1007/s11036-010-0268-0.
URL `http://dx.doi.org/10.1007/s11036-010-0268-0`

[17] H. T. Malazi, K. Zamanifar, S. Dulman, Fed: Fuzzy event detection model for wireless sensor networks, International Journal of Wireless & Mobile Networks (IJWMN) 3 (6) (2011) 29–45.

[18] K. Kapitanova, S. H. Son, K.-D. Kang, Using fuzzy logic for robust event detection in wireless sensor networks, Ad Hoc Networks 10 (4) (2012) 709–722.

[19] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, D. Rubenstein, Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet, SIGARCH Comput. Archit. News 30 (5) (2002) 96–107.

[20] H. Yang, B. Sikdar, A protocol for tracking mobile targets using sensor networks, in: 1st IEEE Int. W. on Sensor Network Protocols and Applications, Anchogare, AK, 2003, pp. 71–81.

[21] C.-Y. Chong, F. Zhao, S. Mori, S. Kumar, Distributed tracking in wireless ad hoc sensor networks, in: 6th International Conference of Information Fusion, Cairns, Queensland, Australia, 2003, pp. 431–438.

[22] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, et al., Envirotrack: Towards an environmental computing paradigm for distributed sensor networks, Int. Conf. on Distributed Computing Systems ICDCS (2004) 582–589.

[23] T. Abdelzaher, J. Han, A survey of datamining methods for sensor network bug diagnosis, in: C. C. Aggarwal (Ed.), Managing and Mining Sensor Data, Springer US, 2013, pp. 429–458.

[24] C.-Y. Lin, W.-C. Peng, Y.-C. Tseng, Efficient in-network moving object tracking in wireless sensor networks, IEEE Transactions on Mobile Computing 5 (8) (2006) 1044–1056.

[25] E. Olule, G. Wang, M. Guo, M. Dong, Rare: An energy-efficient target tracking protocol for wireless sensor networks, in: Int. Conf. on Parallel Processing Workshops ICPPW, Xian, China, 2007, pp. 76–81.

[26] W.-R. Chang, H.-T. Lin, Z.-Z. Cheng, Coda: A continuous object detection and tracking algorithm for wireless ad hoc sensor networks, in:

5th IEEE Consumer Communications and Networking Conference CCNC, Las Vegas, NV, 2008, pp. 168–174.
[27] E. Tanin, S. Chen, J. Tatemura, W.-P. Hsiung, Monitoring moving objects using low frequency snapshots in sensor networks, in: Int. Conf. on Mobile Data Management (MDM), 2008, pp. 25–32.
[28] C. T. Calafate, C. Lino, J.-C. Cano, P. Manzoni, Modeling emergency events to evaluate the performance of time-critical WSNs, in: IEEE Sym. on Computers and Communications, ISCC, Riccione, Italy, 2010, pp. 222–228.
[29] C. Lino, C. T. Calafate, A. Diaz-Ramirez, P. Manzoni, J.-C. Cano, Studying the feasibility of IEEE 802.15.4-based WSNs for gas and fire tracking applications through simulation, in: 11th IEEE Int. W. on Wireless Local Networks (LCN), Bonn, Germany, 2011, pp. 875–881.
[30] V. S. Raja, S. S. S. Mole, A predictive energy-efficient mechanism to support object-tracking sensor networks, IJCA Int. Conf. in Recent trends in Computational Methods, Communication and Controls ICON3C (8) (2012) 13–17.
[31] R. Zhu, Intelligent collaborative event query algorithm in wireless sensor networks, International Journal of Distributed Sensor Networks 2012.
[32] S. Ahn, D. Kim, Proactive context-aware sensor networks, in: Wireless Sensor Networks, Vol. 3868 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 38–53.
[33] Z. Can, M. Demirbas, A survey on in-network querying and tracking services for wireless sensor networks, Ad Hoc Networks 11 (1) (2013) 596 – 610. doi:http://dx.doi.org/10.1016/j.adhoc.2012.08.007.
URL http://www.sciencedirect.com/science/article/pii/S1570870512001540
[34] H. B. Mitchell, Multi-sensor data fusion: an introduction, Springer, 2007.
[35] A. Dallil, M. Oussalah, A. Ouldali, Sensor fusion and target tracking using evidential data association, Sensors Journal, IEEE 13 (1) (2013) 285–293.
[36] R. Kulkarni, A. Forster, G. Venayagamoorthy, Computational intelligence in wireless sensor networks: A survey, Communications Surveys Tutorials, IEEE 13 (1) (2011) 68–96.
[37] G. Shah, M. Bozyigit, D. Aksoy, Adaptive pull-push based event tracking in wireless sensor actor networks, International Journal of Wireless Information Networks 18 (2011) 24–38, 10.1007/s10776-010-0126-9.
[38] N. Malpani, J. L. Welch, N. Vaidya, Leader election algorithms for mobile ad hoc networks, in: 4th Int. W. on Discrete Algorithms and Methods for Mobile Computing and Communications, DIALM, 2000, pp. 96–103.
[39] S. Vasudevan, J. Kurose, D. Towsley, Design and analysis of a leader election algorithm for mobile ad hoc networks, in: 12th IEEE Int. Conf. on Network Protocols, ICNP, 2004, pp. 350–360.
[40] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, Tag: A tiny aggregation service for ad-hoc sensor networks., in: 5th Sym. on Operating System Design and Implementation (OSDI), Boston, Massachusetts, USA, 2002, pp. 131–146.
[41] E. F. Codd, A relational model of data for large shared data banks., Commun. ACM 13 (6) (1970) 377–387.
[42] MEMSIC Powerful Sensing Solutions for a Better Life, http://www.memsic.com/company/about-memsic.html (2010).