University of Pisa

Department of Computer Science

Ph.D. Thesis

Dynamic Detection and Tracking of Composite Events in Wireless Sensor Networks

Claudio Francesco Vairo

Supervisors:
Stefano Chessa and Giuseppe Amato

Pisa, 2012

A mamma,
la mamma migliore del mondo.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Wireless Sensor Networks (WSN) [16] are a recent technology suitable for continuous and unattended monitoring of a large variety of environments. They play an important role in many application fields, including environmental monitoring, disaster area monitoring, structure and people health monitoring, ambient assistant living, home applications, surveillance and security.

Ambient Assisted Living (AAL) [33] is an example of a recent application field for WSN. AAL aims at improving the quality of life of elderly people, by increasing their autonomy, assisting them in their daily activities, and by enabling them to feel included, secure, protected and supported. In AAL applications, WSN are used to monitor the status of the people and environmental parameters of their homes, offices etc. WSN suit very well these applications since they require limited maintenance and they can be adapted and hidden very well in the environment due to their small size.

Another important application of WSN is in the recovery from environmental disasters [58]. In this scenario, WSN can be used to monitor areas that are not accessible by humans and can provide real-time support to emergency management teams operating in the field. In these applications, WSN are appreciated due to their ease of deployment and to the self-organization capability of the sensors.

WSN can also be effectively used in Structure Health Monitoring (SHM) [24]. This application serves as a precaution measure, and it can have great social and economical impact. The goals of SHM systems include detecting, localizing, estimating the extent of the damage and predicting the residual life of the structure. The advantages of using WSN in SHM applications are their low deployment and maintenance cost, their large physical coverage and their high spacial resolution.

A WSN is composed by a (possibly) large number of sensors, that can be easily deployed in the environment (sensing field), and that self-organize to form a (multi-hop) wireless network. The sensors can be programmed to sample parameters of the surrounding environment, to process sampled data, and to forward this information to a special sensor (called *sink*) that provides connectivity with external networks. In typical applications, sensors are tiny microsystems that comprise a

low-performance processor, a limited amount of RAM (few KBs), a set of transducers, and a low-power radio transceiver (in most cases, the radio is compliant with the IEEE 802.15.4 standard [1]). In most cases the sensors are battery powered, although in some cases they make use of energy harvesting techniques to gather energy from the environment [55, 48, 52, 31]. As a consequence, sensors have severe computational and storage constraints, and energy efficiency is critical in most applications. Furthermore, if they are powered by batteries, their lifetime is limited.

Programming WSN to execute specific application dependent tasks is still a relevant issue. In fact, programming sensors requires non trivial skills of embedded systems programming, massively distributed algorithms, and wireless communications. To address this issue, recent proposals [69, 40, 38] suggest the use of database paradigms and declarative languages (generally SQL-like) to specify WSN tasks. In a traditional database system, queries are used to search for data contained in persistent storage repositories. In a WSN, the database consists of the environmental data that can be measured/acquired by the transducers available on the sensors. Queries instruct the sensors on the management, filtering, and processing of the data acquired from the environment. Environmental data are thus acquired by the transducers of the sensors whenever needed, in accordance with the query that the network is processing.

However, the classical approaches to database management systems cannot be applied as such to WSN, where state-of-the-art processing units are characterized by very restrictive resource constraints. Hence most of the aspects related to database systems have to be reinterpreted according to this purpose, and this gave rise in the recent years to a rich research area.

This thesis fits this research trend and provides two main contributions:

- describes the static event detection problem in WSN and presents a solution, the MaD-WiSe system, that efficiently addresses this problem by means of a distributed query processor executed on the sensors of the network. The results of this work are published in [12], [13].

- extends the query processing approach to the mobility context. To this purpose this thesis defines the concept of composite mobile event in a WSN, describes the problem of detecting and tracking such events, and presents an approach, based on a high-level query language, to describe event tracking tasks. Tracking tasks expressed by means of queries can be executed in an automatic and dynamic manner, inside the WSN. The results of this work are published in [62], [11].

1.1 Static Event Detection

Event detection is one of the application fields where WSN are commonly used. In this case the WSN is typically configured in order to continuously collect environ-

mental data, that are sent to the sink where they are analyzed to detect the event. Since the amount of collected data can be considerably large, sending all the data to the sink may result in very inefficient and expensive tasks, especially in terms of energy consumption. This cost can be reduced by executing part of the data analysis and filtering directly on the sensors, thus reducing the communication costs. However, this approach requires programming more complex tasks on the sensors, involving data acquisition, analysis, communication, storage, etc., and this also increases the possibility of introducing programming errors. For this reason, recent proposals introduce abstraction mechanisms for the sensors' hardware, in order to facilitate the programming of distributed applications over the WSN. In particular, in the last few years, several approaches [28, 69, 40, 38] suggest the use of the database paradigm in WSN, and provide high-level SQL-like languages to specify monitoring tasks. With these solutions, the user can thus focus on his needs (*what* he wants from the WSN), rather than concerning on *how* the monitoring task should be implemented. In this thesis we introduce the MaD-WiSe system, our solution to address the problems of efficiently managing data and detecting static events in WSN.

MaD-WiSe is a system for data management in WSN that can efficiently detect static events. It exploits a query language based on SQL, with constructs specialized for WSN, to define the monitoring task. A query defined with this query language expresses the sensing activity to be performed, its timings, and the sensors and transducers involved in it. In support to the query language, MaD-WiSe defines data streams and a query algebra to model the query plan for each sensor. Streams are used to implement both data acquisition and data transfer among sensors, and the query algebra introduces operators that represent aggregation and/or filtering operations on data streams. A query is thus translated into a distributed query plan consisting of operators of the query algebra connected by streams. In order to reduce the energy consumption of the query, MaD-WiSe exploits query optimization strategies and energy efficiency techniques based on the synchronization of the sensors to reduce their periods of activity.

The contributions of this thesis to MaD-WiSe are:

- evaluation of the performance of the system and experimentation of new automatic optimization techniques;
- definition of strategies for the energy efficiency, based on a cross-layer method that extract from the query information about the sampling rate and use this information to synchronize the network and MAC level activities of the sensors;
- a revised query processing model and a redesigned architecture, based on the previous contributions.

MaD-WiSe is implemented in Java and TinyOS [4] for the Crossbow Mica platform [5], and it is presented in Chapter 3.

1.2 Mobile Event Detection and Tracking

Query languages defined to monitor static events in WSN assume that the monitored events do not move, and that they maintain their physical properties (shape, size, etc...) for the whole duration of the query execution (for example, monitoring the temperature in a room, or the health conditions of a building). However there are events that move, or change their size or shape (for example a fire in the wood, or a person moving in the environment). These systems are no longer efficient in tracking this kind of events, since the query needs to be updated to be able to adapt to the changing event. In addition, in traditional query language approaches, the query addresses only specific sources, such as individual transducers on the sensors. On the other hand, the user may be interested in high-level information about the tracked event. For example the user may be interested in the speed and direction of a moving event, rather than on data read from transducers.

In this thesis, we aim at achieving the following goals:

- provide a higher-level of abstraction as compared to the current approaches, for the detection and tracking of events in WSN;
- express queries directly on events, rather than on specific sensors or transducers in the WSN;
- provide a mechanism for an efficient detection and automatic and dynamic tracking of mobile events in WSN.

In the second part of this thesis (Section 4) we enhance the MaD-WiSe approach and we propose a new mechanism to efficiently detect and track mobile events in WSN. To this purpose, we model the concept of composite event in a WSN and we define a new query language, called Event Query language (EQL), for the detection and tracking of mobile events in WSN. Unlike other approaches, with EQL the user can specify a tracking task that, once injected in the network, instructs the sensors on how to cooperatively and autonomously detect and track an event, and on how to dynamically and autonomously migrate the needed query processing tasks in the network as the event moves. We also propose a query processing mechanism for the efficient execution of EQL queries directly in the sensors. We show through simulation that the proposed approach has a lower overhead, and that it scales better with the mobility of the tracked events, as compared to a centralized query approach where the sensors acquire the data and send them to a base station that performs the detection and pilots the tracking of the event.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows. Section 2 presents the state of the art on data management in WSN related to the detection of static events and to

the tracking of moving events. Section 3 is focused on the problem of detecting static events and it describes the MaD-WiSe system, presenting an evaluation of the optimization strategies in MaD-WiSe. Section 4 focuses on the detection and tracking of moving events and it presents the EQL language and the query processing mechanism for the efficient execution of EQL queries. It also includes an evaluation of the proposed system and the comparison with a centralized query approach. Finally, Section 5 draws the conclusions.

Chapter 2

Related Work

2.1 Monitoring of Static Events with WSN

Directed Diffusion [28] was the first attempt to define a data management paradigm in WSN. In Directed Diffusion the user queries the network by injecting interests, associated with a sampling rate, that are broadcasted to all sensors in the network. Data detected by the sensors that match a requested interest are propagated to the sink through the network.

Paradigms integrating database management systems and WSN are the natural evolution of Directed Diffusion. These paradigms use query languages (such as SQL, for instance) to program the sensing task and translate the queries into query plans executable by the sensors. Among the various proposals pursuing this approach, pioneers and (to some extent) state of the art are considered Cougar [69], and TinyDB [40]. The differences between these approaches are in the expressiveness of the query language and on different assumptions on the network architecture.

Cougar [69] is a sensor database that integrates stored data (e.g. characteristics of sensors and their position) and sensor data. Sensor data are physical environment measurements acquired by the sensors, that are sent to the sink as consequence of a query request. In TinyDB [40] all nodes execute locally the same query and the results of the query are merged while they flow from the sensors to the sink. TinyDB can process aggregate queries on data produced by several sensors (spatial aggregation). In [25] TinyDB is extended to support the localization of the sensors, thus enabling the execution of spatial queries, besides the standard ones.

DsWare [72] is another database-like abstraction that is tailored to sensor networks based on event detection. It provides more flexibility by supporting group-based decision, reliable data-centric storage, and implementing a mix of approaches to improve real-time execution performance, reliability of aggregated results and reduce network communication (overhead). DsWare provides applications with services supported by its architecture modules such as data storage, data caching, group management, event detection, data subscription, and scheduling. Like Cougar [69],

DsWare uses SQL-like language for the registration and cancellation of events.

In [38] a PCM-based (probabilistic compensation model) data transmission scheme is proposed to compensate the possible loss of data in executing aggregate queries. An efficient path selection algorithm is also presented, in which children select the most effective parents to send messages, in order to reduce the communication traffic cost while maintaining variance low.

liveSensor [43, 45] is a system that exploits the approach of XML enrichment in order to provide the user with the required semantics to express powerful queries. liveSensor uses XML metadata to describe the data stream, and it provides the user with a high level query interface to deal with XML queries. These queries are transformed from XML to lower level primitives that execute on raw sensor streams.

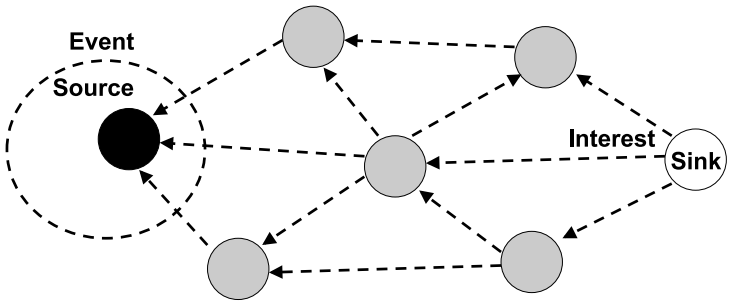
In [75] the authors model the WSN as a distributed database that can be queried to access data generated by the sensor nodes. They develop an algorithm for communication-efficient implementation of joining multiple data streams. In particular, their algorithm is based on the Perpendicular Approach (PA), which is communication-efficient and load-balanced, and has nearly optimal communication cost for binary joins in grid networks under the assumption of uniform generation of tuples across the network. PA works by using appropriately defined horizontal and vertical paths for tuple storage and join-computation, respectively. The approach is able to efficiently incorporate joins with spatial constraints, and can be generalized to sensor networks without location information.

In [17] the authors propose a framework to create and integrate light databases within sensors to manage and process data. The databases are designed by means of simple entity-relationship models from which the code needed to manage the database is automatically generated. This code is composed of data structures and the algorithms in charge of managing them. This architecture helps developers to avoid data redundancy, thus enabling a better management of the sensor's memory, and it contributes to save time in application development.

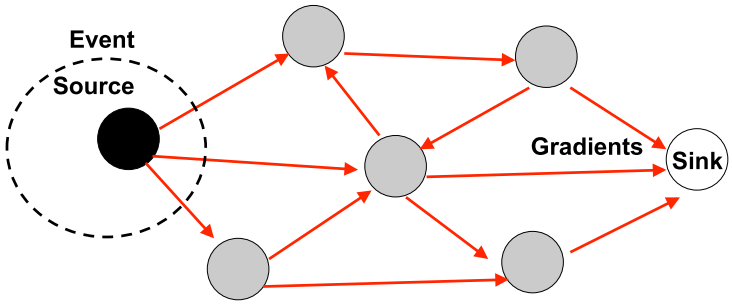
In the next sub-sections we describe in detail those works that are more intimately related to this thesis.

2.1.1 Directed Diffusion

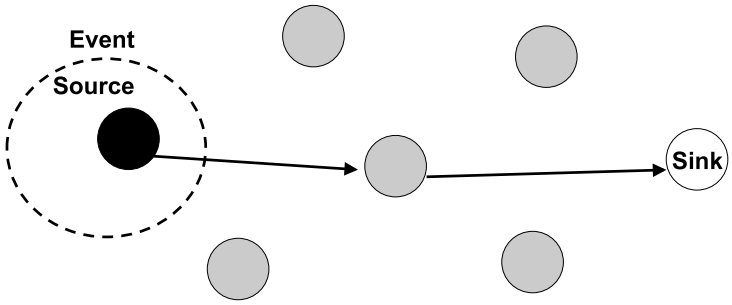
Directed Diffusion [28] proposes a *data-centric* dissemination paradigm for WSN. Data generated by sensors are named by attribute-value pairs. A sensing task is disseminated throughout the sensor network as an *interest* for named data. This dissemination sets up *gradients* within the network designed to "draw" events (i.e., data matching the interest). Events start flowing towards the originators of interests along multiple paths. The sensor network reinforces one, or a small number of these paths. Figure 2.1 illustrates these elements. A detection task that uses the directed diffusion paradigm can be implemented as follows: the user's query is transformed into an interest that is injected in a given region of the network. When a sensor in that region receives the interest, it starts acquiring the information needed to detect



(a) Interest propagation.



(b) Initial gradients set up.



(c) Data delivery along reinforced path.

Figure 2.1: A simplified schematic for directed diffusion.

the event defined in the query. When the sensors detect the requested event, they return the collected information about the event along the reverse path of interest propagation. Intermediate sensors might aggregate the data by combining reports from several sensors.

Naming

In directed diffusion, task descriptions are *named* by a list of attribute-value pairs that describe a task. For example, a task for animal tracking might be described as in Table 2.1.
The task description specifies an interest for data matching the attributes. For this

```
type = four-legged animal     // detect animal location
interval = 20 ms              // send back events every 20 ms
duration = 10 seconds         // ..  for the next 10 seconds
rect = [-100, 100, 200, 400]  // from sensors within rectangle
```

Table 2.1: An example of description of an animal tracking task.

reason, such a task description is called an *interest*. The data sent in response to interests are also named using a similar naming scheme. For example, a sensor that detects an animal might generate the data shown in Table 2.2:

```
type = four-legged animal  // type of animal seen
instance = elephant        // instance of this type
location = [125, 220]      // node location
intensity =0.6             // signal amplitude measure
confidence =0.85           // confidence in the match
timestamp = 01:20:40       // event generation time
```

Table 2.2: An example of data generated by an animal tracking task.

Interests and Gradients

An interest is usually injected into the network at the sink node. Then it is *diffused* through out the sensor network in the following way. For each active task, the sink periodically *broadcasts* an interest message to its neighbors. The initial interest contains the specified `rect` and `duration` attributes, but contains a much larger `interval` attribute (for example, 1 event per second). Intuitively, this initial interest may be thought of as exploratory; it tries to determine if indeed there are any sensor nodes that detect the specified event. The desired data rate is achieved by `reinforcement`. Then, the initial interest may take a form as shown in Table 2.3:

```
type = four-legged animal
interval = 1s
rect = [-100, 200, 200, 400]
timestamp = 01:20:40
expiresAt = 01:30:40
```

Table 2.3: An example of an initial interest of an animal tracking task.

The sink periodically *refreshes* the interest by resending the same interest with a monotonically increasing `timestamp` attribute. This is necessary because interests are not reliably transmitted through-out the network.

Every node maintains an interest cache. Each item in the cache corresponds to a *distinct* interest. Two interests are distinct if their `type` attribute differs, their

`interval` attribute differs, or their `rect` attributes are disjoint. An entry in the interest cache has several fields. A `timestamp` field indicates the timestamp of the last received matching interest. The interest entry also contains several `gradient` fields, up to one per neighbor. Each gradient contains a `data rate` field requested by the specified neighbor, derived from the `interval` attribute of the interest. It also contains a *duration* field, derived from the `timestamp` and `expiresAt` attributes of the interest, indicating the approximate lifetime of the interest. When a node receives an interest, it checks if the interest exists in the cache. If no matching entry exists, the node creates an interest entry. The parameters of the interest entry are instantiated from the received interest. This entry has a single gradient towards the neighbor from which the interest was received, with the specified event data rate. In the example above, a neighbor of the sink sets up an interest entry with a gradient of 1 event per second towards the sink. If there exists an interest entry, but no gradient for the sender of the interest, the node adds a gradient with the specified value. It also updates the entry's `timestamp` and `duration` fields appropriately. Finally, if there exists both an entry and a gradient, the node simply updates the `timestamp` and `duration` fields. When a gradient expires, it is removed from its interest entry.

After receiving an interest, a node may decide to send again the interest to some subset of its neighbors. To its neighbors, this interest appears to originate from the sending node, although it might have come from a distant sink. In this manner, interests *diffuse* throughout the network. Not all received interests are sent more than once. A node may suppress a received interest if it recently re-sent a matching interest.

Data Propagation

A sensor that detects a target searches its interest cache for a matching interest entry. In this case, a matching entry is one whose `rect` encompasses the sensor location, and the `type` of the entry matches the detected target type. When it finds one, it computes the highest requested event rate among all its outgoing gradients. The node activates its transducers and generates event samples at this highest data rate. In the example, this data rate is initially 1 event per second. Then, every second, the source sends a *data* message to each neighbor for whom it has a gradient, containing an event description, as shown in Table 2.4:

A node that receives a data message from its neighbors attempts to find a matching interest entry in its cache. If no match exists, the data message is dropped. If a match exists, the node checks the *data cache* associated with the matching interest entry. This cache keeps track of recently seen data items to prevent loops. If a received data message has a matching data cache entry, the data message is dropped. Otherwise, the received message is added to the data cache and the data message is re-sent to the node's neighbors.

```
type = four-legged animal  // type of animal seen
instance = elephant        // instance of this type
location = [125, 220]      // node location
intensity =0.6             // signal amplitude measure
confidence =0.85           // confidence in the match
timestamp = 01:20:40       // event generation time
```

Table 2.4: A data message containing an event description in response to an animal tracking task.

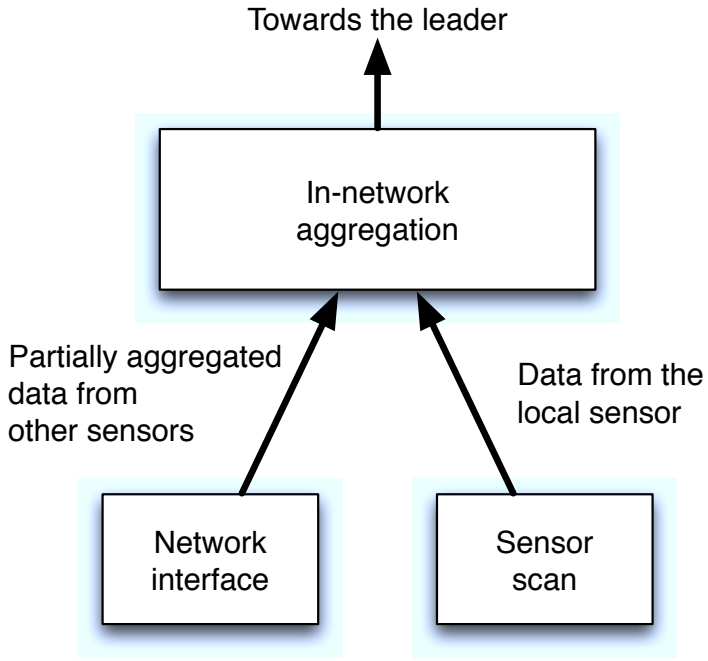


Figure 2.2: Cougar Query Plan at a source sensor.

Reinforcement

After the sink starts receiving the low data rate events, it *reinforces* its neighbors in order to "draw down" higher quality (higher data rate) events. To reinforce a neighbor, the sink resends the original interest message but with a smaller `interval`, for example $10ms$. When a neighboring node receives this interest, it notices that it already has a gradient towards this neighbor, and that the sender's interest specifies a higher data rate. If this new data rate is also higher than that of any existing gradient, the node also reinforces its neighbors. Through this sequence of local interactions, a path is established from source to sink for the transmission of high data rate events.

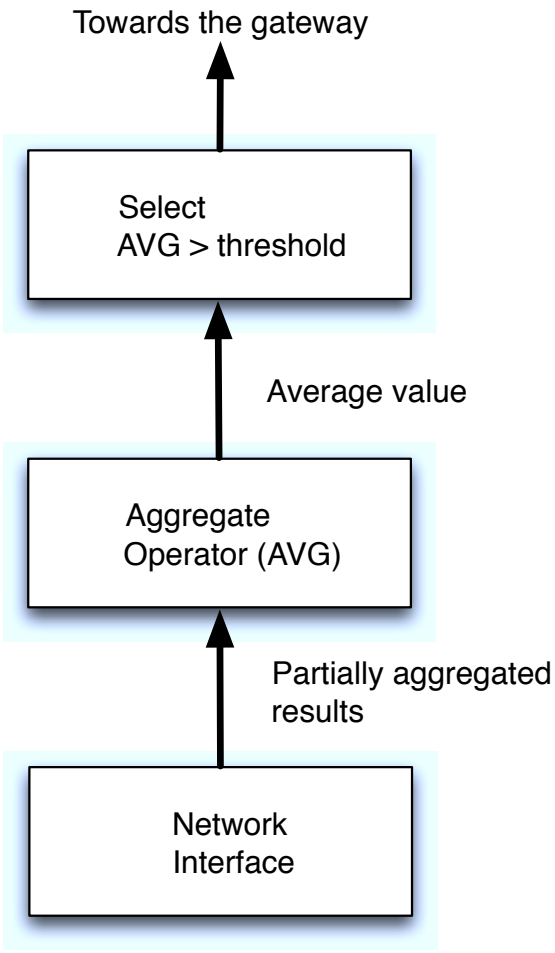


Figure 2.3: Cougar Query Plan at the leader.

2.1.2 Cougar

Cougar [69] is one of the first works integrating the database approach to WSN. To enable declarative querying of sensor networks, Cougar proposes a *query layer* consisting of a *query proxy* running on every sensor node. The Cougar architecture on the sensors is composed of three layers: the network layer, the query proxy layer and the application layer. The query proxy provides higher-level services through queries that can be injected into the network from a specified gateway node. In order to reduce the energy consumption, one of the main roles of the query proxy is to perform in-network processing.

A query optimizer is located on the gateway node to generate distributed query processing plans after receiving queries from the outside. The query plan is created according to catalog information and the query specification. Such a query plan specifies both the data flow (between sensors) and an exact computation plan (at

each sensor). The plan is then disseminated to all relevant sensor nodes. Control structures are created to synchronize sensor behavior, and the query is started. At run-time, data records flow back to the gateway node as in-network computation happens on-the-fly.

In order to illustrate the individual components of the architecture in more detail, consider the following example: a long-running query Q monitoring the average temperature of an office every t seconds. The query Q has to notify the user (i.e., Q generates an output record) if the average temperature in the office is greater than t. As a first step in evaluating the query, the query optimizer generates a query plan QP that specifies how to determine the *leader* of this query, a designated node where the computation of the average temperature will take place. The leader could be a fixed sensor with more remaining power and energy, or a randomly selected node by some distributed leader election algorithm. Two computation plans are then produced, one for the leader node, and a second plan for the remaining nodes in the query region.

Figure 2.2 shows the query plan for a non-leader node that participates in the query. Non-leader nodes have a scan operator to sample transducers readings periodically and to send them to the leader node. In addition, their plan contains an aggregation operator to aggregate data from other sensors. Figure 2.3 shows the query plan for the leader node, which contains an AVG operator to compute the average value over all sensor readings received in the last round of the query, and a SELECT operator that checks if the result is above the threshold.

At query start time, the generated query plans are disseminated to the query proxies of all relevant sensor nodes. The query proxies registers the query, create a local operator tree, active relevant transducers, and return records according to the specification of the query plan. The leader generates a record only if the average temperature is above the user-defined threshold.

2.1.3 TinyDB

TinyDB [40] is a distributed acquisitional query processor for data collection in sensor networks. By focusing on the locations and costs of acquiring data, TinyDB allows to reduce power consumption compared to traditional passive systems. TinyDB operates at all levels of query processing: in query optimization, in query dissemination, and in query execution. It also inherits a lot of features and optimizations of a traditional query processor (e.g., the ability to select, join, project, and aggregate data). Figure 2.4 illustrates the basic architecture of TinyDB: user submits a query at the sink, where the query is parsed, optimized, and disseminated in the WSN. In turn, the sensors process the query and send the results back to the sink, through the routing tree that was formed during the query dissemination.

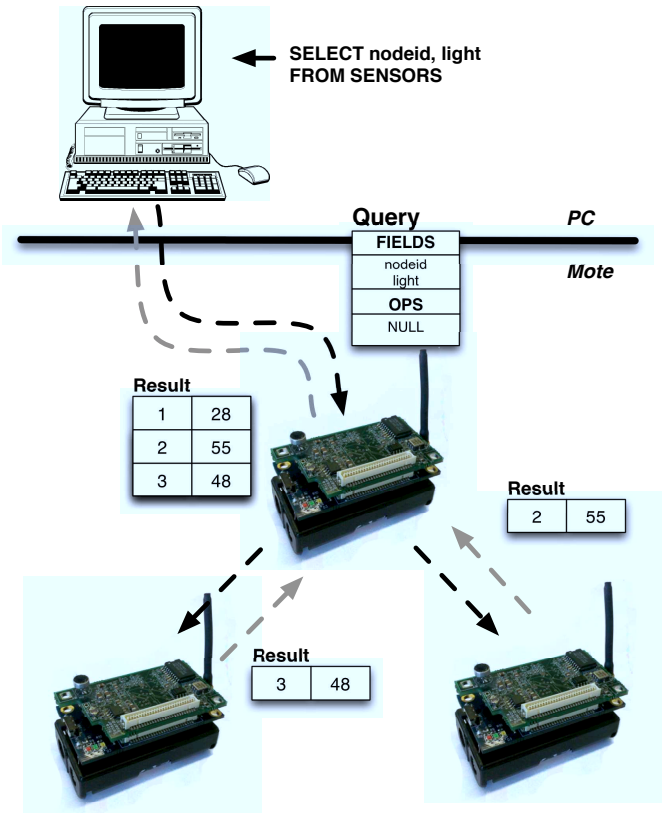


Figure 2.4: A query and results propagating through the network.§

Data Model

In TinyDB, sensor tuples belong to a table `sensors` which, logically, has one row per node per instant of time, with one column per attribute (e.g., light, temperature, etc.) that the device can produce. Records in this table are put (i.e., acquired) only as needed to satisfy the query, and are usually stored only for a short period of time or delivered directly to a terminal outside the network. Sensors can insert `NULLs` for attributes corresponding to missing transducers. Physically, the `sensors` table is partitioned across all of the devices in the network, with each device producing and storing its own readings. Thus, to compare readings from different sensors, those readings must be collected at some common node, for example, the sink.

Basic Language Features

Queries in TinyDB consist of a `SELECT-FROM-WHERE-GROUPBY` clause supporting selection, join, projection, and aggregation, just as in SQL, with the same semantic. Tuples are produced at well-defined *sample intervals* that are a parameter of the query. The period of time between two consecutive sample periods is called *epoch*. An example of TinyDB query is shown in Table 2.5:

```
SELECT nodeid, light, temp
FROM sensors
SAMPLE PERIOD 1s
FOR 10s
```

Table 2.5: An example of TinyDB query.

A sensor executing the query in the table returns (i.e. it put the data information in the virtual table `sensors`) its own identifier, light, and temperature readings once per second for 10 s. The output consists of a stream of tuples, clustered into 1-s time intervals. Each tuple includes a time stamp corresponding to the time it was produced. When a query is issued in TinyDB, it is assigned an id that is returned to the issuer and it is used to identify data related to a particular query.

TinyDB also includes support for grouped aggregation queries: as data from an aggregation query flows up the tree, it is aggregated in-network according to the aggregation function and value-based partitioning specified in the query. However it cannot execute queries that relate and compare in-network data acquired by different sensors (for instance, to check if the temperature in room 1 is greater than that in room 2).

Event-Based Queries

TinyDB supports *events* as a mechanism for initiating data collection. Events in TinyDB are generated explicitly, either by another query or by a lower-level part of the operating system (in which case the code that generates the event must have been compiled into the sensor node).

```
ON EVENT bird-detect(loc):
SELECT AVG(light), AVG(temp), event.loc
FROM sensors AS s
WHERE dist(s.loc, event.loc) < 10m
SAMPLE PERIOD 2s
FOR 30s
```

Table 2.6: An example of event-based query in TinyDB.

For example, the query in Table 2.6 reports the average light and temperature level at sensors near a bird nest where a bird has just been detected. Every time a `bird-detect` event occurs, the query is issued from the detecting node and the average light and temperature are collected from nearby nodes once every 2 s for 30 s. In this case, it is assumed that bird-detection is done via some low-level operating system facility, such as a switch that is triggered when a bird enters its nest.

Metadata Management

Before queries are disseminated, the sink performs a simple query optimization phase to choose the ordering of sampling, selections, and joins that minimizes the power consumption. Each node in TinyDB maintains a catalog of metadata that describes its local attributes, events, and user-defined functions. This metadata is periodically copied to the sink to be used by the optimizer. Event metadata consists of a name, a signature, and a frequency estimate that is used in query optimization. User-defined predicates also have a name and a signature, along with a selectivity estimate which is provided by the user himself. Metadata associated with node attributes are: how much energy and time are required to sample this attribute, if this attribute is constant or variable, what range this attribute can take on, how fast this attribute can change. The catalog also includes names of aggregates and pointers to their code. Each aggregate consists of a triplet of functions, that initialize, merge, and update the final value of partial aggregate records as they flow through the system. TinyDB also stores metadata information about the costs of processing and delivering data.

Power-Based Query Optimization

Sampling is a more expensive operation, in terms of energy consumption, than executing an operator. However, if a predicate is required to be evaluated over an attribute of a sensor node, the corresponding transducer must be activated to execute the sampling. On the other hand, if a predicate discards a tuple of the `sensors` table, then subsequent predicates do not need to examine the tuple, so the cost of sampling any attributes referenced in the subsequent predicates can be avoided. Thus, ordering carefully the expensive operators, can save energy. In order to efficiently order the predicates, TinyDB considers two problems: (a) an attribute may be referenced in multiple predicates, and (b) expensive predicates are only on a single table, `sensors`. The first point introduces some subtlety, as it is not clear which predicate should be "charged" with the cost of the sample.

To model this issue, TinyDB handles the sampling of a sensor as a particular operator, τ, that is scheduled along with the predicates, p. TinyDB deals the problem of determining the minimum-cost order of predicates and sampling operations, under the constraint that τ_i must precede p_j if p_j references the attribute sampled by τ_i. The proposed solution is as follows: (a) data are acquired only when any predicate over that data must be evaluated, unless it was already sampled in order to evaluate a previous predicate; (b) if a query requires two or more sampling operations, they are performed in ascending order of sampling energy.

Dissemination and Routing

After the query has been optimized, it is disseminated into the network; dissemination begins with a broadcast of the query from the sink. As a node receives the query, it decides whether the query applies locally and/or whether it needs to be

broadcast to its children in the routing tree. We say a query q applies to a node n if there is a nonzero probability that n will produce results for q. If a query does not apply at a particular node, and the node does not have any children for which the query applies, then the entire subtree rooted at that node can be excluded from the query, saving the costs of disseminating, executing, and forwarding results for that query. In order to address the problem of determining when a node or its children need not participate in a particular query, TinyDB uses a data structure called a semantic routing tree (SRT), that maintains information about child attribute values. With SRT a node can determine whether none of its children currently satisfy the value of some selection predicate, for example, because they have constant (and known) attribute values outside the predicate's range.

Conceptually, an SRT is an index that can be used to locate nodes that have data relevant to the query. When a query q with a predicate over a constant attribute A arrives at a node n, n checks if any of the values related to A and produced by its children overlap the query range of A in q. In this case, it prepares to receive results and forwards the query. Otherwise, the query is not forwarded. Moreover, if the query also applies locally, n begins executing the query itself. If the query does not apply at n or at any of its children, it is simply ignored. Building an SRT is a two-phase process: first the *SRT build request* is flooded through out the whole network. This request specifies the name of the attribute A over which the tree should be built. When a node n receives the request, if it has no children, then it chooses a node p from available parents to be its parent, and reports the value of A to p in a *parent selection message*. If n has children, it forwards the request to them and waits for their replies. When it has heard from all of its children, it chooses a parent and sends a selection message indicating the range of values of A which it and its descendants cover. Because children can fail, nodes also have a timeout which is the maximum time they will wait to hear from a child; after this period is elapsed, the child is removed from the child list. Figure 2.5 shows an SRT over the X coordinate of each node on a Cartesian grid. The query arrives at the root and it is forwarded down the tree. Only the gray nodes are required to participate in the query (node 3 must forward results for node 4, despite the fact that its own location precludes it from participation).

Processing Queries

Once queries have been disseminated and optimized, the query processor executes them. Time is divided in epochs. Nodes sleep for most of an epoch to minimize power consumption. Once a node is awake, it begins sampling and filtering results according to the plan provided by the optimizer. Filters are applied and results are routed to join and aggregation operators further up the query plan. Results are put into a radio queue for delivery to the node's parent. This queue contains both tuples from the local node, as well as tuples that are being forwarded on behalf of other nodes in the network. The queue can fill depending on the number of queries

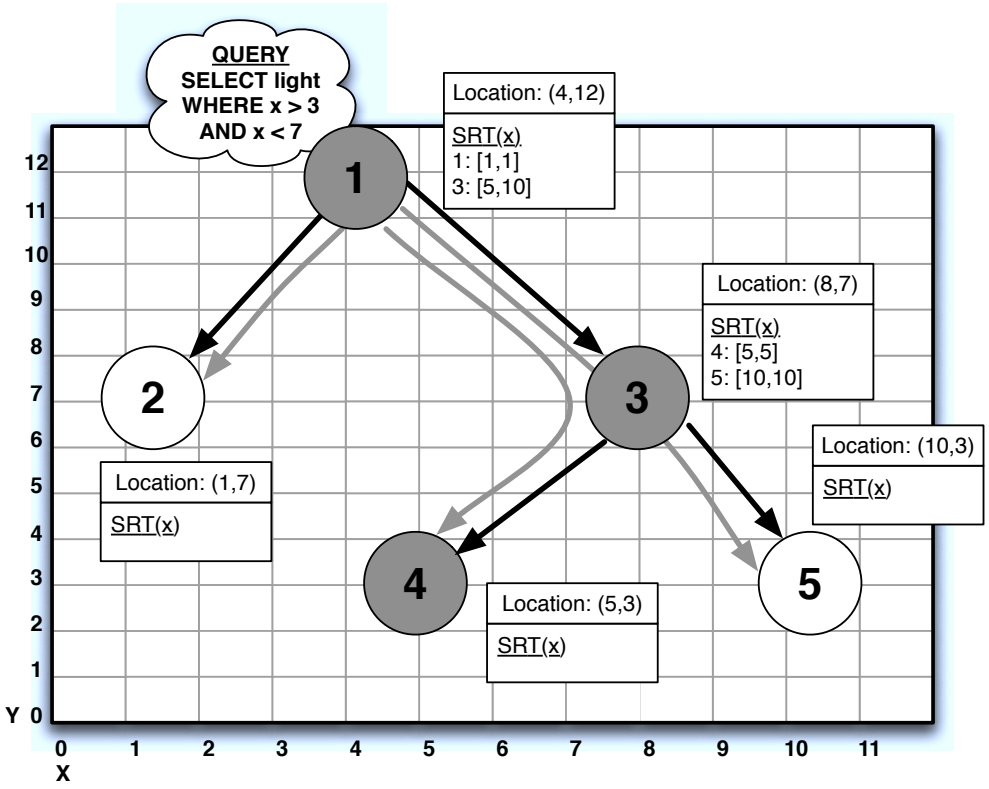


Figure 2.5: A semantic routing tree in use for a query. Gray arrows indicate flow of the query down the tree; gray nodes must produce or forward results in the query.

running, the cardinality of joins, and the number of groups and aggregates. In this case, the system must decide if it should discard the overflow tuple, discard some other tuple already in the queue, or combine two tuples via some aggregation policy.

2.2 Detection and Tracking of Mobile Events with WSN

Kumar et al. [50] propose a framework for distributed event detection where groups of nodes cooperate to detect a composite event. The framework consists of two protocols that build a tree to detect an event by using a communication model similar to the Publish-Subscribe paradigm, where the communication is done in an asynchronous manner. An application subscribes to an event by specifying the area where the event is expected to occur. Then the protocol builds an event-based data collection tree. For composite events, a counter is maintained for each atomic event part of the composite event. Counters keep track of the number of sensors which can sense atomic events. Sensors are added to the tree until counters exceed some

predefined thresholds.

In [64] the Timely Energy-efficient k-Watching Event Detection problem (TEKWED) is examined. The authors move the computation needed to detect an event inside the network, which now takes place at a gateway node. This node is responsible for reaching a conclusion and notifying the users when an event happens. The gateway node is properly selected, and every sensor in the network has a chance to serve as a gateway node in order to balance the energy consumptions. The sensors are grouped in detection sets and only one detection set is active at any time, thus saving energy. The gateway node disseminates the threshold to be checked to detect the event to the sensors, and they 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.

In [35] the authors propose a cluster based, energy-aware event-detection scheme where events are reliably relayed to a sink in the form of aggregated data packets. The clustering scheme proposed provides fast and accurate event detection and reliability control capabilities to the areas of the network where an event is occurring. In particular, 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 proposed scheme includes an energy-level based CH selection that ensures that higher energy nodes remain as cluster heads for a 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.

In [73] the authors investigate the detection performance of randomly deployed WSN. They address the problem of detecting mobile targets that have continuous movement. They distinguish two kind of targets. In the first type, called rational targets, the knowledge of existing sensors is assumed. In the second type of targets this assumption does not hold, and it is assumed that the targets neither know anything about the sensors, nor do they plan their path for specific purposes, while their traces are assumed to be straight lines. They propose a mechanism to find critical positions to deploy additional sensors, so that the freedom of mobile targets can be limited, and the probability of detection on mobile targets is increased.

In [68] the Scheduling for Composite Event Detection (SCED) problem is addressed. They propose a sensor scheduling mechanism that ensures both coverage and connectivity requirements, and maximizes the network lifetime. They assume that sensors are densely deployed, and that they are equipped with multiple sensing components for watching a composite event. The proposed method is to put some redundant sensors to sleep. They consider the network lifetime organized in rounds, where each round has two phases: initialization phase and data collection phase. The initialization phase chooses a set of active sensors as sensing nodes in order to achieve both coverage and connectivity requirements. Sensors which are not chosen to be active in this round go to sleep to save energy. In the data collection phase, active sensors perform sensing and data relaying.

In [74] the authors investigate the problem of how to design a detection scheme to minimize the energy consumption of WSN while providing the required detection accuracy. They propose an adaptive scheme that leverages the discrepancy among individual sensor's detection accuracy, which is obtained from a collaborated training process, to allow each sensor to operate at its most energy efficient level while guaranteeing the overall detection accuracy.

Recently, the fuzzy logic paradigm has been applied in the event detection in WSN. FED [41] is a fuzzy event detection model that benefits from fuzzy variables to measure the intensity as well as 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 [32] the authors focus on improving the definition of the event to be detected in order to increase the detection accuracy. In particular they believe 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.

One of the first works on tracking moving objects with WSN is ZebraNet [30]. In this work the sensors operate as a peer-to-peer network to compute local information about the moving objects to be tracked. This information is then stored inside the network. The base station periodically traverses the network and gathers the acquired data from the sensors when it is close to them.

In [67] a distributed and scalable prediction-based algorithm that accurately tracks mobile targets using WSN 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 prediction-based approach is also used in [21], where the cluster organization of the network is exploited to track multiple moving objects.

EnviroTrack [6] 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. The 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 which 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. With network management mechanisms

such as lightweight group management and group leader election, it supports the dynamic behavior of the tracked targets such as mobility.

In [36] 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 tree updating and query execution costs.

RARE [47] proposes two algorithms for performing an efficient target tracking based on a static cluster organization of the WSN. The first algorithm reduces the number of sensors participating in tracking by excluding the sensors too far from the target object. The second one reduces redundant information by identifying overlapping sensors.

In [8] the data acquired by the sensors are analyzed, in a centralized manner, outside the network, in order to detect phenomena to track and from which data is acquired. A phenomenon is detected when different sensors report similar readings over a period of time. Once the phenomenon has been detected, the information about the spatial properties of the phenomenon are used to optimize subsequent user queries.

DELTA [66] is a fully distributed event localization and object tracking framework for WSN. DELTA adopts a distributed algorithm for detecting an event and for dynamically building groups around the object to be tracked. For each group, a leader is responsible for group maintenance, gathering and processing of tracking data. In [65] the DELTA framework is extended with mechanisms for the localization and classification of events.

CODA [20] is an algorithm for the detection and the tracking of continuous phenomena (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 detecting 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.

In [59] the main effort is in reducing the number of environmental readings and, as a consequence, the amount of data delivered outside the network. They take infrequent snapshots of the area and they adopt a low-quality target tracking algorithm to maintain object identities. The target tracking algorithm they adopt is a leader-based algorithm, in which the node that is considered to have the best reading for the moving object, is elected as leader. 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 base station.

In [18] the authors present a novel framework for time-critical event generation in WSN environments 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 visualizing the different events generated. In [37] they use the framework developed in [18] to performed 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 [53] 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.

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

In the next sub-sections we describe in more detail the works that are more related to this thesis about detection and tracking of mobile events.

2.2.1 Distributed Cooperation for Event Detection

Kumar et al. in [50] propose a framework that exploits the collaboration among sensors for distributed event detection in WSN. It consists of two protocols (*simple event detection* protocol and *composite event detection* protocol) that build a tree to detect an event. Each protocol is composed of two phases, namely *initialization phase* and *collection phase*. Different components are present on different nodes depending on the functionality of each node. The communication model that is used in the framework is similar to the Publish/ Subscribe paradigm [60]. It is scalable and it offers the advantage to entirely decouple the publishers from the subscribers so the communication is done in an asynchronous manner.

In the initialization phase, the application submits events of interest to the sensor network. Here the application acts as a subscriber for the event of interest. The events of interest can be a simple event or a combination of simple events. Based on these events, an *event based tree* (EBT) is built according to the Publish/Subscribe communication paradigm. Nodes which can generate data related to any simple event join the tree. Generation of data depends on the sensing capabilities of the nodes, thus, an EBT is constructed as required by the application. This tree can be shared by different applications if a simple event is present in more than one events of interest supplied by them. The collection phase is responsible of gathering results after an event has occurred. The aggregation happens at each node depending on the event and the corresponding generated data.

An application submits predicates over attributes in the form of $A_1 > value_1, A_2 >$

$value_2, ..., A_n > value_n$, where $A_1, A_2, ..., A_n$ are low level sensing attributes like temperature, pressure, etc. Subscription messages with the above mentioned predicates are generated by the middleware, that also plays the role of event disseminator in the WSN. After receiving a subscription message, each node checks whether it has the capability to sense the attributes specified in the message. If so, it produces an entry of the application identifier (*application id*) along with the *parent id*, that is the node from which it has received the message. The application id is maintained to distinguish between different subscriptions. A node receiving a subscription from a user node will act as the sink for that subscription, and the related tree is built with the sink as root.

The application submits the event to be reported in the form of a simple event. It also specifies the location regions where the event has to be detected, the number of hops, and the tolerable delay between events. The number of hops determines the number of groups to be formed in the desired region. This parameter is entirely dependent on the application. For example, in a target tracking application sensors will form several groups in the region of interest to increase the probability of detection.

In this work, the sensors are assumed to be aware of their position. Thus each node can check whether it is present in the desired region and, hence, decide whether to participate in the event detection task. The system keeps track of how many nodes are required to infer that an event i has happened, and it also keeps track of how many nodes are currently detecting that particular event. If the number of nodes which are already detecting some event exceeds the minimum number of nodes requested to detect that event, the other nodes can skip detecting this particular event, hence saving energy, or they may contribute to detect other events. In this way nodes collaborate among themselves and contribute for event detection.

2.2.2 Composite Event Detection

Vu et al. [64] propose a scheme for detection of composite events in WSN. They move the computation needed to determine the occurrence of the event, inside the network, in one or more *gateway nodes*. The network is composed of different types of sensors. For example, in Figure 2.6, *type1*, *type2*, and *type3* sensors are used for temperature, smoke density, and light monitoring, respectively. An event E is defined as following:

$$E = \mathcal{F}(P_1(x), ..., P_n(x)) \tag{2.1}$$

where $P_1(x)$ through $P_n(x)$ are predicates, and $\mathcal{F}$ is a function of Boolean algebra operators such as '$\wedge$', '$\vee$', or '$\neg$'. For example, an event *fire* can be defined as $Fire = P_1(x) \wedge P_2(x) \wedge P_3(x)$, where $P_1(x)$ denotes the predicate $temperature >$

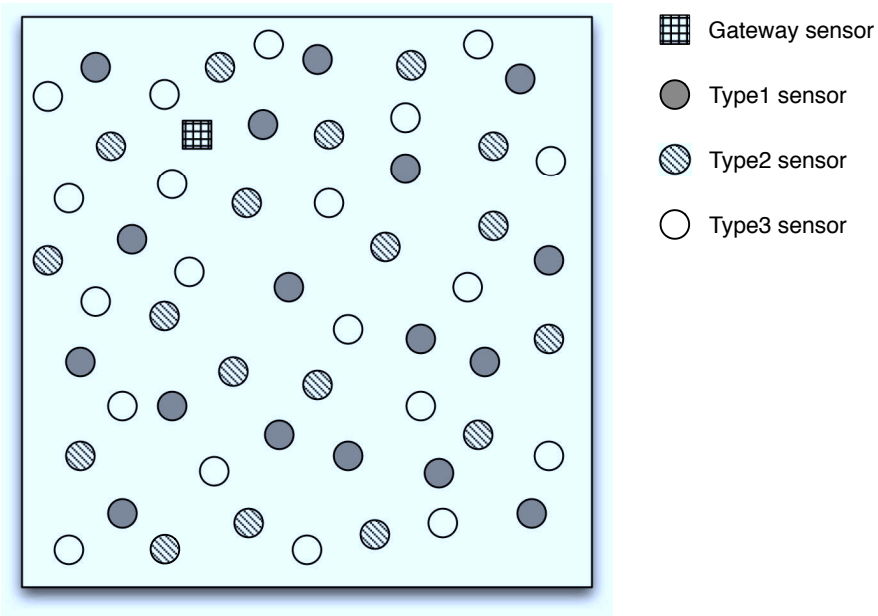


Figure 2.6: Network architecture in Composite Event Detection.

300°C, $P_2(x)$ denotes the predicate $smoke > 100mg/L$, and $P_3(x)$ denotes the predicate $light > 500cd$.

In event alarming applications the connectivity is extremely important since an event needs to be alerted on time. However, energy conservation is always a primary objective for WSNs. To meet these requirements, the proposed scheme divides the sensors into non-disjoint subsets (*detection sets*), and each subset can conduct the event alarming process with a user defined fault-tolerance level. Instead of requiring all the sensors to be active all the time, only one subset is responsible for the event alarming task at any time. In this way, energy can be conserved and the network lifetime can be extended. Each detection set is provided with a BFS tree built at the set definition moment; such tree is used to route the messages to the gateway node. During the active time of a detection set, the sensors in this set route their messages to the gateway node based on the topology and routing information provided by the sink.

The detection set construction algorithm first selects a gateway node (where the gateway can simply be any node with enough residual energy). Then, it constructs a set of connected BFS-like trees rooted at the gateways, and the nodes in each of these trees form a detection set. Several parameters are considered by a sensor in order to be added to a tree: its residual energy, the energy needed to transmit a message to its parent node, and the number of transducers that belong to the predicates of the event to detect. Once the detection sets are built, the sensors start monitoring the events to be detected.

An event E and the related threshold values can be either disseminated to gate-

way and non-gateway nodes by the sink at the initial phase, or pre-installed in each sensor. Only the gateway nodes have the information about an event E. All the other nodes only know the threshold values of its monitored properties. During the network operation time, when a sensor detects that the current sensed value is over the threshold of its monitored property, it sends one bit '1', instead of the sensed value, to the gateway node in its detection set. If a gateway node receives a '1', it checks if the propositional function which defines the event E derives a $TRUE$ value. If so, it immediately notifies the sink. By sending only one bit of notification, instead of sending raw data each time, the amount of data transmitted in the network is significantly reduced.

2.2.3 ZebraNet

ZebraNet [30] is a WSN system for wildlife tracking. In this case, the sensors are tracking collars carried by animals across a large area. The collars are equipped with GPS, and they operate as a peer-to-peer network to deliver logged data back to the user. The scenario in which operates ZebraNet is the following:

1. sensors store their readings (GPS position included) for 3 minutes every hour. Differently from traditional WSN applications, they do not send the acquired data to the user immediately;

2. the WSN runs for 1 year without the human intervention;

3. the sensing-field is very large (over 40,000 square Km);

4. the sink is not fixed; it periodically traverses the network to gather the stored data;

5. latency is not critical, but a high success rate (close to 100%) for the data collection phase is required;

The collars are equipped with two radios: a data radio with a range of only 100 meters but a very low power consumption, and a slow but higher-power data radio for longer-range (8km) transmissions. The short-range radio is power-efficient for peer transmissions, while the longer-range radio is necessary for communicating to the sink to deliver the data. However, even with the long-range radio, the area where the sensors operate is too large for direct communications with the sink. Sensors forward data coming from other sensors in multi-hop paths. In ZebraNet the sensors are extremely mobile, therefore no fixed network topology can be built to collect data. Also the sink is mobile, depending on the route taken by researchers in their vehicles. Furthermore, it is active only for a limited period of time, when researchers are driving around gathering data. During the period when the sink is inactive, the network has no known destination where data can be sent.

A simple approach to send data to the sink is to flood data to all neighbors whenever they are discovered. If the sensors move extensively and meet a fair number of other sensors, data will eventually reach the sink. In this way, the percentage of the delivered data is high. By using multi-hop paths, the sink does not need to interact directly with all the sensors in the system. While flooding can potentially return the highest success rate in a peer-to-peer network, the large amount of data flooded through the network can lead, in some situations, to exorbitant demands for network bandwidth, storage capacity, and energy.

ZebraNet also considers a history-based protocol that smartly selects sensors that will be visited by the sink to collect data, based on prior communication patterns. A good target sensor is one that can relay the data to the sink. This protocol encodes the likelihood of a sensor being within the sink's range by assigning each sensor a hierarchy level, based on its past success at transferring data to the sink. The higher the level, the higher the probability that this sensor is within sink range. The idea behind this approach is that sensors that were previously within range of the sink will still be close by, so they will be able to relay the data back to the sink either directly (if they are still within range), or indirectly through a reduced set of intermediate sensors. Each sensor remembers its own current hierarchy level. Each time a sensor scans for neighbors, it requests the hierarchy level of all of its neighbors. It then sends the data it has collected to the neighbor with the highest hierarchy level, with ties randomly broken. When a sensor comes within range of the sink, its hierarchy level is increased. Conversely, when a sensor is outside the range of the sink for a long time, its hierarchy level becomes lower at a rate of one level per every D scans, where D is a configuration parameter. At the beginning, all sensors start with a hierarchy level equal to zero. The success of the history-based unicast routing protocol depends on the degree of mobility of both sink and sensors. If the network changes very dynamically, a sensor that was previously near the base station may no longer be the best communication target. Then, the history-based protocol may result in a low success rate.

Chapter 3

Monitoring Static Events with MaD-WiSe

The detection of static events is one of the most common tasks in WSN. This task is often achieved by instructing the sensors to continuously collect environmental data and to send them to the sink for analysis. To implement this task efficiently, the most common approach [69, 40, 13, 7] is to instruct the sensors to perform a preliminary filter and preprocessing of the data, in order to reduce the communication costs. As observed in many of the works presented in Chapter 2, the tasks of data sensing, filtering and preprocessing can be easily expressed by means of SQL-like queries. In this chapter we present MaD-WiSe, a system for data management in WSN that exploits a query language based on SQL to define a monitoring task. MaD-WiSe defines a stream based model and a query algebra to represent the query plan for each sensor, and it uses a query optimization strategy to reduce the energy consumption of the query. More specifically:

- MaD-WiSe defines a *query language* with constructs specialized for WSN. The query language manipulates data sources consisting of specific transducers located on individual sensors. The queries thus express the sensors and transducers involved in the sensing activity and the timings of such activity. Queries can relate and compare data acquired by multiple (remote) sensors, and they can also aggregate data in the spatial and temporal dimension. Queries are fully executed in the sensors of the WSN. Section 3.2 discusses these aspects.

- To support the query language MaD-WiSe defines a *stream model* for WSN. Streams are used to represent both data acquisition and data transfer among sensors, and the model defines different energy costs depending on the stream nature. The stream model is presented in Section 3.3.1.

- MaD-WiSe defines a *query algebra* that introduces operators that represent aggregation and/or filtering operations on data streams. A query is thus translated into a distributed query plan consisting of operators of the query algebra

connected by streams. The query is executed by the WSN in a distributed fashion: each sensor involved in a query executes the part of the query plan assigned to it. The query algebra is described in Section 3.3.2.

- MaD-WiSe uses an algebraic approach to *query optimization* that uses transformation rules based on heuristics aimed at reducing the energy consumption of the query. Query optimization issues are discussed in Section 3.6.

It should be observed that the main purpose of MaD-WiSe is different than that of traditional databases, where the main issue is to process queries as quickly as possible. In fact, MaD-WiSe is oriented at the WSN lifetime maximization, that implies a minimization of the energy consumed by queries. Besides query optimization, MaD-WiSe adopts energy efficiency techniques based on the synchronization of the sensors to reduce their periods of activity and communications. To this purpose, the MaD-WiSe query optimizer extracts information about the data sampling rate from the query, and uses this information to configure the MAC and network layers of the sensors. Regarding this issue, MaD-WiSe exploits a cross-layer optimization.

The MaD-WiSe architecture comprises two main components, one component, called *client-side*, that runs on the sink (in our case a PC or a PDA), and one component, called *network-side*, that runs on the sensors. Both components are organized as a protocol stack composed of independent layers. The MaD-WiSe architecture is described in Section 3.4

MaD-Wise is implemented in Java and TinyOS [4] for the Crossbow Mica platform [5] and it can be downloaded from the MaD-WiSe project web site [3].

Note that an early version of MaD-WiSe was already implemented at the beginning of this thesis. However, that preliminary work was greatly improved during the development of this thesis under these respects:

- we have first analyzed the performance of MaD-WiSe by means of simulations and experimentations to identify and assess new query optimization techniques;
- we have introduced cross-layer optimization strategies that extract information about sampling rate of the query, and we have used this information to configure the network and MAC layers of the sensors;
- we have revised the query processing model according to the above points, and we have redesigned the architecture to implement the new optimizations and the energy efficiency strategies.

3.1 Design Goals and Assumptions

The MaD-WiSe is a complex system, that addresses several aspects related to the efficient data management in a WSN. It has been designed keeping in mind the following goals:

- **Modularity**: despite the limited computational and memory resources of the sensors, the network-side component of MaD-WiSe is not monolithic. It is organized as a protocol stack with three layers with well defined interfaces. This enables the possibility of replacing the policies implemented at each level without affecting the whole system.

- **Distributed in-network query processing**: MaD-Wise aims at carrying the major part of the computation within the network. To this purpose the sensors execute the queries cooperatively and the query language exploits join operators in order to enable in-network aggregation and filtering of data produced by different sensors.

- **Abstraction of the Wireless Sensor Network**: the user submits high-level queries and he/she does not have to be concerned on how the results are actually collected and processed.

- **Hybrid push/pull behavior**: the system provides both on-demand and periodic data acquisition.

- **Energy efficiency**: MaD-WiSe adjusts the energy consumption to the effective needs of the query, by exploiting cross-layer techniques to synchronize the sensors.

- **Optimization**: MaD-WiSe exploits optimization techniques to compute the query execution strategy that are able to reduce the energy consumption.

Consistently with several other similar approaches [70, 40, 54, 61], MaD-Wise also assumes a static network topology for the whole duration of an injected query. Furthermore, it assumes that the client-side component of MaD-WiSe knows the static locations of the sensors. This knowledge can be obtained from the network itself, using techniques as those proposed in [19], or it can be manually generated when the network is deployed, for instance, by annotating on a map the location where the sensors were placed in the buildings or in the field subject to the monitoring application. The current implementation of MaD-WiSe adopts the latter method. In particular, it uses a manually-generated configuration file that specifies the set of available sensors, their positions, and their sensing capabilities (i.e. the set of on-board transducers).

3.2 Query Language

The MaD-WiSe query language, called MW-SQL, shares its basic constructs with SQL. MW-SQL allows users to express queries to manipulate, filter, and organize sequences of tuples generated by the sensors. While SQL uses the concept of *table*, to denote a typology of data stored, MW-SQL relies on the concept of *source* to

provide the user with an abstraction of a sequence of tuples arriving from a precise origin. MW-SQL queries are expressed through query statements whose complete syntax is reported in Section 3.2.1 (additional details can also be found in the MaD-WiSe manual [29]). The simplest type of data source (called *basic source*) is an individual transducer on a sensor. MW-SQL also handles *complex sources* that are constructed by combining together other sources (basic or complex) by means of the MW-SQL operators. The operators include timestamp join (that is a join on the timestamp attribute), spatial aggregation, union, etc. and they are described in Section 3.3.2.

To get the flavor of the MW-SQL language, consider the following MW-SQL query:

SELECT *
FROM avg(1.Temperature, 2.Temperature, 3.Temperature) as AVG
WHERE AVG.Temperature $>$ 25
EVERY 10000

This query requests the WSN to compute the average of the values received every 10 seconds from the basic sources corresponding to the temperature transducers on sensors 1, 2, and 3 (the aggregate of the temperature measured by the three sensors is a *spatial average*, that is an aggregate computed between the values of the same fields in different sources). If the average temperature is above 25, it is sent to the sink. The query involves three sensors, each of which executes different operators connected by streams. The information passed from one operator to the other consists of tuples, each containing a data and its timestamp. Hence, in this case, the result of the query is a pair $(TS, Temperature)$, where TS is the timestamp and $Temperature$ is the computed average value. The $Temperature$ parameter in the **FROM** clause is different from the one in the **WHERE** clause.

By means of clause **EVERY**, the user states at which frequency the query should be executed (in this particular case the user requests data to be delivered every 10 seconds). This information is extracted from the query by the Query Parser, and it is given, along with the query plan, to sensors 1, 2, and 3 that are involved in the query. Once these sensors begin to execute the query, they activate the transducers and the radio interface every 10 seconds, only for the time needed to perform the samplings and the communications required.

MW-SQL can also express temporal aggregates, that is, aggregates computed between the values acquired in a given time interval. This can be expressed by specifying the aggregate operator in the **SELECT** clause, and by defining the time interval using the **EPOCH** clause. Other examples of queries can be found in Table 3.1 and Table 3.3.

If the sensors' coordinates are known (for instance, if they embed GPS or if they are assigned some sort of virtual coordinate [19]), it is possible to choose the sources for a query by specifying the area where the needed sensors are located by indicating the top-left and bottom-right corners of a rectangular area.

3.2.1 MW-SQL Syntax

MW-SQL specifies two constructs. One construct is used to express queries. The other one is used to create virtual sources.

A MW-SQL query is defined as:

query::= **SELECT** *select-list*
FROM *source*
[**WHERE** *conditions*]
[(**EPOCH**|**WINDOW**) *samples* [**SAMPLES**]]
[**EVERY** *rate*]

A select list is a list of fields or (temporal) operators on fields:

select_list::=*select_element*(,*select_element*)*|**
select_element::=*t_aggr*(*field*) | *field*
t_aggr::=**MAX**|**MIN**|**AVG**|...
field::=*simple_source*.*field_name*
field_name::=*string*

The from clause takes as argument a source. There are various types of sources:

source::= *single_source*|*source_list*
single_source::=*named_s_s*|*unnamed_s_s*
named_s_s::=*unnamed_s_s* **AS** *virtual_source*
unnamed_s_s::=*simple_source*|*basic_source*|*s_aggr_source*| **UNION**(*source_list*)
|*query*
simple_source::=*virtual_source*|*node_source*
virtual_source::=*string*
node_source::=*number*|*area_source*|**ALL**
basic_source::=*node_source*.*transducer_name*
transducer_name::=*string*
s_aggr_source::=*s_aggr*(*source_list*)
area_source::=**AREA**(*number*,*number*,*number*,*number*)
source_list::=*single_source*(,*single_source*)+
s_aggr::=**MAX**|**MIN**|**AVG**|...

Conditions take the following form:

conditions::=*condition* [**AND** *condition*]*
condition::= *exp op exp*
exp::= *field*|*number*
op::= =| < | > | <= | >= |...

Finally the timing is expressed as:

samples::=*number*
rate::=*number*

The construct for creating virtual sources has the following syntax:

virt-source::= **CREATE SOURCE** *source-name*
AS *source*

source-name::=string

3.3 Query Processing Model

A WSN can be seen as a *distributed* data stream managements system, where data can be accessed, processed, and cross-related using distributed query processing techniques [34, 14, 46].

In MaD-WiSe the tasks executed by the WSN are expressed by using SQL-like queries. A query is translated by the sink node (in case of MaD-WiSe a standard PC that is connected to the WSN and that runs the client-side module of the MaD-WiSe system) into a query plan that consists of a set of operators (defined in the query algebra) connected by streams. Streams connect operators executed either on the same sensor or on different sensors. Streams are also used to connect data sources (i.e. transducers) to operators. The query is first handled by the Query Parser (running on the sink), which generates a query execution plan, that, in turn, is transformed by the Query Optimizer (also running on the sink) into an equivalent, energy-efficient query plan. To this purpose the optimizer uses the configuration file that specifies the set of available sensors and their capabilities. Finally, the sink injects the optimized query plan into the sensors that, in turn, execute the query. The operators of the query algebra are executed exclusively on the sensors. Therefore, once a query is submitted, the WSN is able to autonomously execute it. Note that in MaD-WiSe data are not stored into the sensors. Instead the portion of the query plan they are executing specifies when the data should be acquired from the environment, how it should be processed, and where it should be sent. In the following we describe in more detail the query language, the streams, and the query algebra operators of the MaD-WiSe system.

3.3.1 Streams

MW-SQL queries are parsed at the client side and translated into a query execution plan consisting of operators of the query algebra (discussed in Section 3.3.2)

connected by streams of tuples. MaDWiSe uses three types of streams to provide operators with three modes of accessing data: *sensor streams*, which represent streams of data acquired by the transducers, *remote streams* that model streams of data connecting two operators on different sensors, and *local streams* that represent streams of data generated by the execution of local operators and sent as input to other local operators. These three types of streams model data acquisition (sensor streams), data transmission (remote streams) and pipelined processing on a single sensor (local streams). The flow of tuples in a stream is modeled by a single *dynamically changing tuple t*. A stream maintains the last received tuple, and every new tuple overwrites the previous one, forcing an on-the-fly processing of the tuples. A small buffer can also be used to store tuples in case of delays during execution of operators.

In the following the three types of streams are described.

Sensor Streams are used to sample environmental data. A sensor stream receives tuples of data acquired by the associated transducer. Tuples in a sensor stream have the following structure: $(TS = ts, NI = ni, TR = value)$, where TS is the field containing the timestamp, NI is the field containing the identifier of the node, and TR is the name of the field corresponding to the transducer used to acquire a value (for instance, Temperature, Light, Acceleration, etc.). Three different modes for receiving the tuple of a sensor stream are considered:

1. In the *periodic update* mode the transducer is activated to acquire a tuple at a fixed rate. The time interval between two consecutive transducer activations is called *sampling period*.

2. In the *on-demand update* mode the transducer is activated as a consequence of a read request on the stream that causes a tuple to be acquired. This mode can be used to obtain transducers readings only under specific conditions.

3. In the *asynchronous activation* mode the tuple is acquired when some external asynchronous event occurs, like, for instance, when a button is pressed.

Periodic updates can be used for periodic monitoring, for example to collect the temperature readings every 10 seconds. On-demand updates can be used by the Query Processor to acquire a value only when some other condition is verified (for instance the light values are read only when the temperature is above a given threshold). Asynchronous updates can be used to detect asynchronous environmental events (for example a vehicle passing through a gate). The update mode of a sensor stream is decided during the query plan generation and query optimization, on the basis of the role of the stream in the query. In on-demand sensor streams the transducer activation (and then the tuple acquisition) is executed (on-demand) when the read operation is invoked.

Remote streams. In general the queries are processed cooperatively by a group of sensors. In these cases it is necessary partial query results produced by a sensor to

be sent to another operator located on another sensor, in order to complete the query execution. For this purpose remote streams are used, where source and destination endpoints (operators) are located on different sensors.

Local Streams model data transfers between operators located on the same sensor in order to enable pipelined executions of the operations. Local streams differ from remote streams since they have different costs in terms of energy consumption. Local streams mainly consume memory resources, while their energy consumption is negligible. On the other hand, remote streams mainly consume energy due to the transmission of the tuples, and this energy consumption is predominant for the cost evaluation of a query execution plan.

3.3.2 Operators of the query algebra

Operators of the query algebra read data from one or more streams and, after processing the read tuples, write (output) result tuples on another stream. An instance of an operator can be seen as a thread that runs on a specific node of the WSN. All operators are non-blocking and have a strictly pipelined behavior: as soon as an operator reads a tuple it processes the tuple and, if needed, it immediately writes the result in the output stream, avoiding the use of temporary buffers for producing results.

Regarding the *selection* and the *projection* of the tuples, MaD-WiSe provides two basic operators that have the same semantics of the corresponding operators of the traditional relational algebra.

In addition, MaD-WiSe provides a special definition for the *n-ary* operators, like *spatial aggregation*, *union*, and *join*, that require the interaction of several sensors. These operators are implemented as a combination of *binary operators* composed to form a binary query execution tree (see Figure 3.1), using an approach similar to that proposed in [15, 40] for computing aggregates. The nodes of the tree are operators that compute partial states of the *n-ary* operator. To this purpose they use locally computed data and/or partial states computed by their children. The nodes are allocated and executed by different sensors of the network; this means that, depending on their allocation, the nodes are connected by local or remote streams. The representation of the partial state of binary operators depends on the specific operator being executed. In some cases the partial state has a size comparable to that of the final operation result (for instance in the case of max and min), while in other cases the size of the partial state might be larger and can raise issues related to resource scarcity (as in case of the median).

Clearly, given a group of sensors, an *n-ary* operator can be computed by organizing and connecting the binary operators in different way. However, the way in which they are connected affects the cost of execution of the whole *n-ary* operator. In section 3.6 we will discuss how the Query Optimizer can produce a query plan consisting of a communication tree that minimizes the energy consumption.

In the following we discuss the *n-ary* operators in more detail.

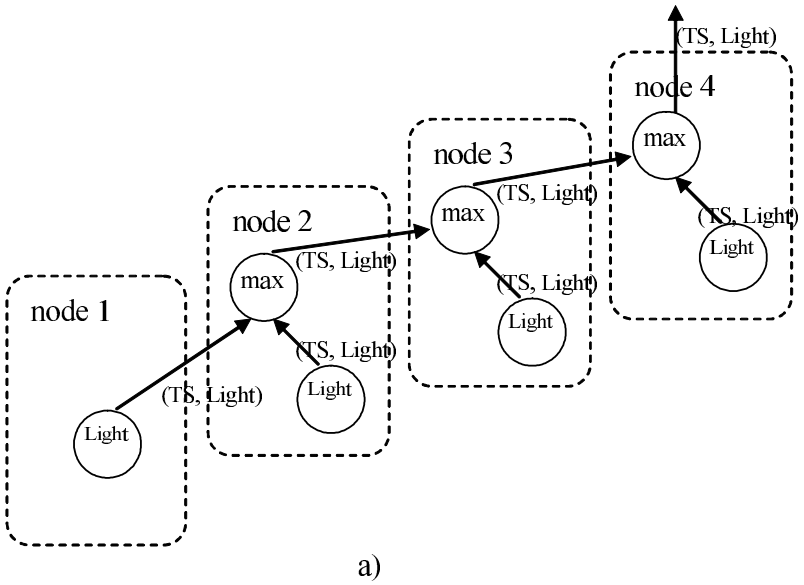
Spatial Aggregation aggregates data produced by different sensors, which might be located in different positions of the space (field) where the WSN is deployed. For example, consider applications that compute the average, the maximum, or the minimum of a given parameter measured at different locations of an environment. Aggregation of data produced by a group of sensors is a very significant feature in WSN since it allows reducing the amount of data that is sent to the sink.

Figure 3.1(a) shows an example of how the maximum light measured by a group of sensors can be obtained. Every node computes the maximum between the value acquired locally and the maximum computed by its child. The computed aggregate value is passed to the parent node. Figure 3.1(b) shows an example where the average light measured by a group of sensors is computed. In this case, the partial state is represented by the sum of the values to be aggregated and the number of values that have been summed. The average is computed in the root of the tree by dividing the sum of the values by the number of values. This is achieved using two operators: partial average (*pavg*) and final average (*favg*). The former one computes the sum and counts the values, and it is used in the intermediate nodes of the binary tree. The latter one is used at the root, and it divides the sum by the number of values. The *pavg* operator produces a tuple of type (*TS*, *Light*, *count*), where the *Light* attribute contains the sum of the light readings, and *count* contains the number of values that were summed. The *favg* operator produces a tuple of type (*TS*, *Light*), where the *Light* attribute contains the computed average. Other aggregates might need other types of partial state information to be transferred across the nodes of the tree.

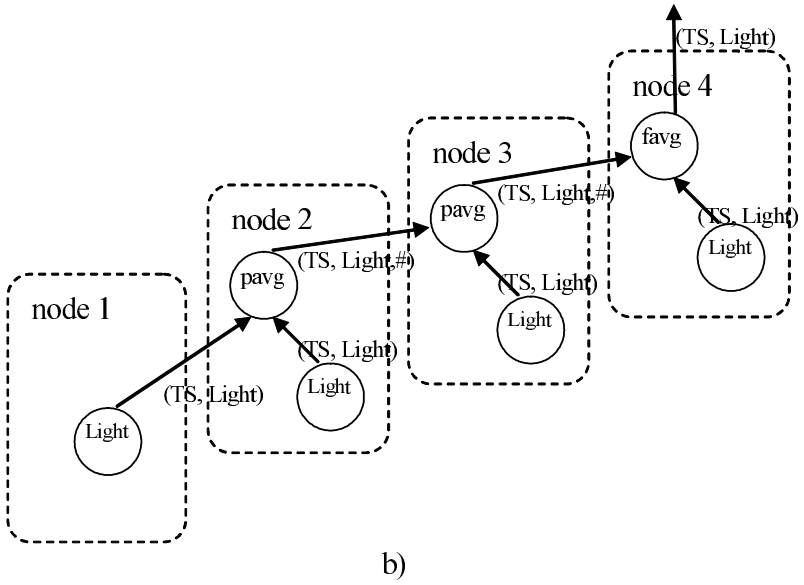
Union is used to gather data from several homogeneous basic sources, possibly located on different nodes. In this case the partial nodes of the query execution tree acquire the local data and add it to the current query result message. The root node simply forwards all the received data to the next operator.

MaD-WiSe provides two definitions for the **Join operator**: *timestamp-join* and *sync-join*. The *timestamp-join* operator, $\bowtie (S_1, S_2)$, is the default join operator in MaD-WiSe, and it relates data acquired (almost) simultaneously on both input streams. When a new tuple arrives on either stream, the *timestamp-join* operator checks whether the last tuple received on the other stream has the same timestamp. In this case it writes the tuple obtained by their combination in the output stream. This operation is clearly non-blocking, and its execution requires only single-position buffers. To solve the synchronization problem among sensors, a low resolution timestamp is used. In this way, even if two values are not read (strictly) simultaneously, they have the same timestamp.

The definition of the *timestamp-join* operator implies that the two input streams are read independently and in parallel. If several tuples (with different timestamps) arrive at the first input stream before a tuple arrives at the second input stream, all the readings from the first stream, except the last one, will be useless. This produces a useless energy consumption due to the transducers or radio activations.



(a) Spatial maximum.



(b) Spatial average.

Figure 3.1: Example of query execution plans for the spatial maximum (a) and for the spatial average (b). Spatial average uses two different operators: *pavg* (partial average) and *favg* (final average)

Consider, for example, the query in Table 3.1. It retrieves the light and temperature readings only when the temperature is above the specified threshold[1]. By

[1]Differently than standard SQL, where the FROM clause indicates a cartesian product among

```
SELECT *
FROM Room1.Light, Room2.Temperature
WHERE Room2.Temperature > 20
EVERY 10000
```

Table 3.1: An example of a query executing a timestamp Join in MW-SQL. This is different from standard SQL, where the FROM clause indicates a cartesian product, since here the FROM clause indicates a timestamp join among sources by default.

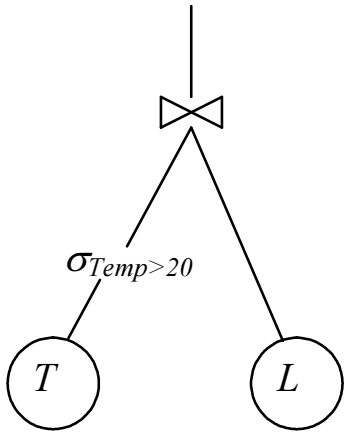


Figure 3.2: Using the join operator, both sensor streams T and L should be periodic. Using the sync-join, the L sensor stream can be on-demand, i.e. it is activated only when needed

the previous join definition, this query can be processed by using two *periodic* sensor streams associated with Light L and temperature T, respectively, and by using the query execution plan given in Figure 3.2. Temperature and Light transducers are both activated every 10 seconds. When the temperature is below the specified threshold, no temperature tuple is sent to the join operator, and the current light reading is lost, since it cannot be matched. Activating the light transducers when the temperature is below the threshold is useless, and it introduces an additional energy consumption. If the probability that the temperature is above the specified threshold is very low, the above query plan consumes a lot of energy for unnecessary light readings.

The same query can be processed more efficiently by defining L as an *on-demand* sensor stream and by using a special definition of the join operator that requests the activation of L only when a tuple arrives on the other stream. This avoids useless transducer activations on the on-demand stream, thus saving energy. This operator is called *sync-join*, $\bowtie_{sync}(S_1, S_2)$, where S_2 is an on-demand stream. The *sync-join* operator combines both the push and the pull sensing techniques and has a master-

tables, in MW-SQL the FROM clause indicates timestamp join among sources.

slave behavior: stream S_1 is the master and the slave stream S_2 is read only when a tuple is received from the master. In this case, the sensor in which the slave stream is instantiated, has to be the sensor that executes the join operator.

In addition to the operators described so far, MaD-WiSe also provides a **Temporal Aggregation** operator that aggregates data over a time interval. Temporal aggregation is achieved by using *epochs* (that is, jumping windows), or *moving windows*. Epochs are consecutive, non overlapping, fixed size time intervals associated with individual queries. Every epoch contains all tuples whose timestamp falls within the corresponding interval. More formally, let us assume that the evaluation of a query starts at time t_0, and that the epoch duration for that query is t_e. The i-th epoch of that query is the interval $Ep_i = (t^i_{start}, t^i_{end}]$, where $t^i_{start} = t_0 + i \cdot t_e$ and $t^i_{end} = t_0 + (i+1) \cdot t_e$. For moving windows, each time a tuple arrives, the oldest one is deleted and the new one is stored in the sensor.

The temporal aggregation operator that uses epochs, groups together tuples belonging to the same current i-th epoch, which is determined by using t_0, t_e, and the epoch counter i. When the epoch ends, the operator computes the final aggregates and deposits the result in the output stream. Then it computes the first partial state for the next epoch. The output tuple generated at the end of the epoch contains the computed aggregated value and the timestamp set to the end of the epoch. The functions used to compute partial and final states, and the information maintained in partial states, depend on the specific aggregation operator being used. For instance, for aggregations like the median, the partial state may contain all readings made during an epoch.

Instead, the temporal aggregation operator that uses moving windows, groups together tuples belonging to the same moving window. Partial states are continuously updated when a new tuple comes. The final aggregation is also computed every time, after the initial delay needed to fill the time window.

The complete semantics of the MW-SQL operators can be found in [9].

3.4 Architecture

The MaD-WiSe system consists of a set of modules that implement a distributed stream management system on a WSN. Some of the MaD-WiSe modules (network side) run on the sensors of the WSN and others (client side) run on a PC or on a PDA connected to the WSN through a special sensor called *sink*. See Figure 3.3 for an illustration.

3.4.1 Client-Side

The client side sub-system is composed of a user interface, a Query Parser, an execution plan Optimizer, and a Query Manager. The user interface allows the user to instruct the sensor network, by submitting MW-SQL queries, and to visualize

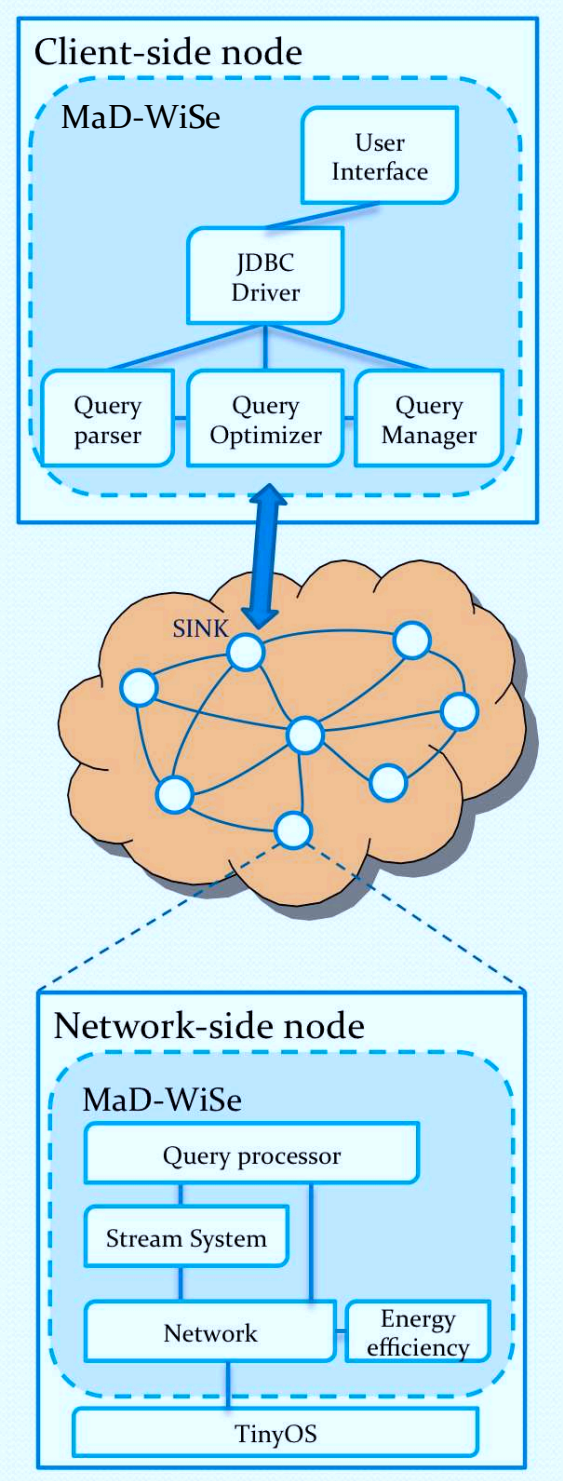


Figure 3.3: The architecture of MaD-WiSe.

Radio Mode	Power Consumption
Idle	0.016 mW
Ready	12.36 mW
Receive (Rx)	12.50 mW
Transmission (Tx)	
0.1 power level	12.36 mW
0.4 power level	15.54 mW
0.7 power level	17.76 mW

Table 3.2: Power Consumption of the radio of an IRIS mote.

the results received. The Query Parser takes the MW-SQL query and translates it into an initial distributed query execution plan, that is, it generates the operators that execute the query and a first allocation plan of the operators on the sensors. The Query Optimizer then generates a semantically equivalent query execution plan where the operations to be executed and the assignments of operators to sensors of the WSN are reorganized so that the costs required for radio communication and transducer activations are reduced. The Query Manager disseminates the optimized query execution plan in the network and handles the results received during the in-network query execution.

Client applications that use MaD-WiSe can be built by relying on JDBC (Java DataBase Connectivity) [44], using a JDBC driver developed for MaD-WiSe. This way, WSN applications can be almost entirely developed using standard database tools.

3.4.2 Network-Side

The network side of MaD-WiSe is developed in nesC [26] on top of the standard MAC layer of TinyOS [4] for the sensor platform MicaZ/Iris [5]. Every sensor contains three software layers (Network, Stream System, and Query Processor), as depicted in Figure 3.3. The layers interact through well defined interfaces, and are autonomous with respect to each other. Each layer can be replaced with a new (different) implementation provided that it complies with the existing interfaces.

Network Layer

In MaD-WiSe the network layer should support routing between arbitrary pairs of sensors. There are several implemented protocols providing this service in WSN, such as, for example, [2] or [27]. The current version of MaD-WiSe uses a simplified version of [49].

The Network layer offers both connectionless and connection-oriented communication services. In the second case, a reserved path is established between the sensors by means of the *connect* command, and it is used for the communications

that follow a query. The network layer also implements an energy efficiency strategy for the connection-oriented service. This strategy manages the duty cycles of the sensors, in particular the on/off periods of the radio interfaces according to the sampling rate expressed in the queries. In fact, the sampling rate of a query defines the rate at which fresh data are sent by the sensors to the sink through the channels implementing the remote streams of the query. The energy efficiency strategy of MaD-WiSe is described in more detail in Section 3.4.3

Stream System Layer

The Stream System layer offers abstraction mechanisms for data access by means of data streams. It can be seen as the equivalent of a file system on a sensor network, the main difference being that, in the former, data is dynamically produced, rather than stored, as a consequence of acquisition from transducers and communication between sensors. The Stream System implements the three types of streams (the sensor, local and remote streams) described in Section 3.3.1. The Stream System interacts with the network layer by means of an interface that defines the *connect*, *disconnect* and *send* commands. The interface also provides two events, *connectDone* and *receive* that notifies to the Stream System layer the completion of the *connect* procedure and the receipt of a message, respectively. When the Stream System layer invokes a *connect* command with a specified destination, the network layer finds a route (called channel) to the destination and allocates the needed data structures on the sensors involved in the channel. When the connection is established, the network layer notifies the completion of the operation by signaling to the Stream System layer a *connectDone* event. The *send* command takes as input the message to be sent and the identifier of the channel to be used for the communication, and sends the message over this channel. Upon receiving a message, the Stream System layer is notified of the incoming message by means of the *receive* event, specifying the identifier of the channel the message is received from.

Query Processor Layer

The Query Processor layer is the core component of the system. It implements a Query Processor of a distributed data stream management system over the Stream System. It offers the implementation of the operators of the query algebra discussed in Section 3.3.2, and it orchestrates the execution of distributed query plans composed of operators connected by streams. The Query Processor of every sensor can be programmed by the client-side subsystem in order to take part in a distributed query execution. In particular, it is instructed by means of messages sent over the serial, in case of the sink, and over the wireless channel, in case of all other sensors. These messages contain information about the operator that has to be executed on the sensor, and about the streams that have to be opened. As a consequence, the Query Processor allocates the data structures related to the operator to be executed.

The Query Processor also interacts with the Stream System to open the required streams, by means of an interface that provides the following commands:

- *open_s, open_l, open_r*: opens the specified stream: a sensor stream in the case of *open_s*, a local stream in the case of *open_l*, and a remote stream in the case of *open_r*. All these commands return the stream descriptor that is used by the Query Processor to use the stream. The commands *open_s* and *open_r* take as input the sampling rate at which the data has to be acquired or sent, respectively.

- *close* is used to remove the specified stream.

- *read* allows the Query Processor to read the data from the specified stream. If no data is available on the stream the *read* operation terminates without returning any data. However, as soon as a new data arrives on the stream, the Query Processor is notified by means of the *readDone* event.

- *write* is used to insert data into the specified stream.

3.4.3 Energy Efficiency in MaD-WiSe

The purpose of the energy efficiency strategy is to let the sensors keep their radios off whenever possible, i.e. when they do not expect to send or receive data. In fact, as observed in other works [71, 57], the radio interface is one of the main sources of energy consumption, and the most critical to optimize. In general, the sensor's radios have four states: idle, ready, Tx (transmission mode) and Rx (receive mode). The energy consumption is very low in idle mode, and it is high for ready, Tx or Rx modes, as shown in Table 3.2. Furthermore, the energy consumption in Tx, Rx and ready modes is quite similar. Hence, to save energy, sensors should keep the radio idle as much as possible. On the other hand, while in idle mode, the radio cannot receive and send packets, and the sensors in this mode behave as if they are disconnected from the network. Hence, if the radio of the sensors is not properly switched off, the network may get disconnected and the sensors may become unable to communicate. Existing MAC layer approaches [10, 23, 51] exploit this feature to reduce the radio energy consumption. We build the energy efficiency strategy of MaD-WiSe based on the work [10].

This strategy manages the on/off periods of the radio interface according to the duty cycles associated to the queries the sensor is involved in. More precisely, each *connect* message has, as a parameter, the sampling rate at which the data are requested by the sink. This information is given to all the energy efficiency components of the sensors involved in the path supporting the connection. In turn, all these components control the radio activations accordingly for the entire duration of the query.

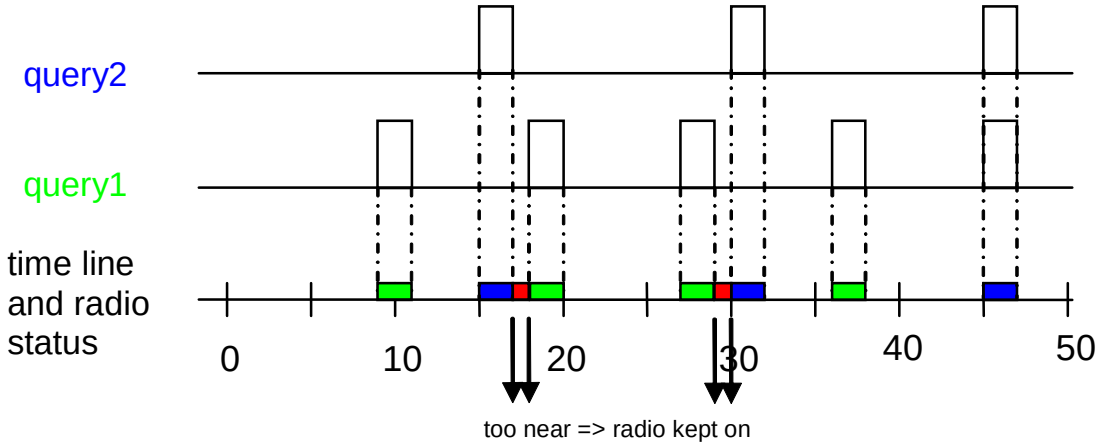


Figure 3.4: Energy efficiency mechanism.

Figure 3.4 shows an example of how the energy efficiency mechanism works, by showing the status of the radio of a sensor involved in the execution of two queries (query1 and query2). The first two lines show the duty cycles related to the two queries. The last line shows the overall radio activity (portions of this line not marked by a colored bold line, mean radio turned off). In this example query1 has a duration of 2 seconds, and a period of 9 seconds, while query2 has a duration of 2 seconds and a period of 15 seconds. The energy efficiency component computes the union of all the duty cycles of the sensor, and determines when it should turn off the radio, according to two additional parameters, *tolerance* and *radioDelay*, that provide more flexibility to the system. The *tolerance* parameter specifies the minimum time interval between the end of a duty cycle and the beginning of the following one: if two duty cycles are too close the radio is kept on until the second one ends (as happens in Figure 3.4 at second 17 and at second 29). The logic behind this behavior is that turning the radio on and off consumes additional energy and time, so two commutations of the radio could waste more energy than keeping the radio on. The *radioDelay* parameter is used to postpone the radio turning off, so that the delay introduced by the transmission of the packets over a multi-hop path is compensated. The duration of the duty cycles is set such that a packet can traverse the longest path in the network and reach the destination before the end of the duty cycle.

The performance of the energy efficiency mechanism is assessed by measuring the periods of radio activity of a sensor. In this evaluation, we use 4 IRIS motes [5] (s1, s2, s3, s4) connected in a line, i.e. s1 is connected to s2, s2 to s3 and s3 to s4. Node s1 acts as the sink, and receives the results of the queries, and the queries are all directed to s4, which is the node that produces data. In the experiments, we measure the radio activity on node s2. We repeat four sets of experiments with a number of queries ranging from 0 to 4. The rate of each query is set randomly

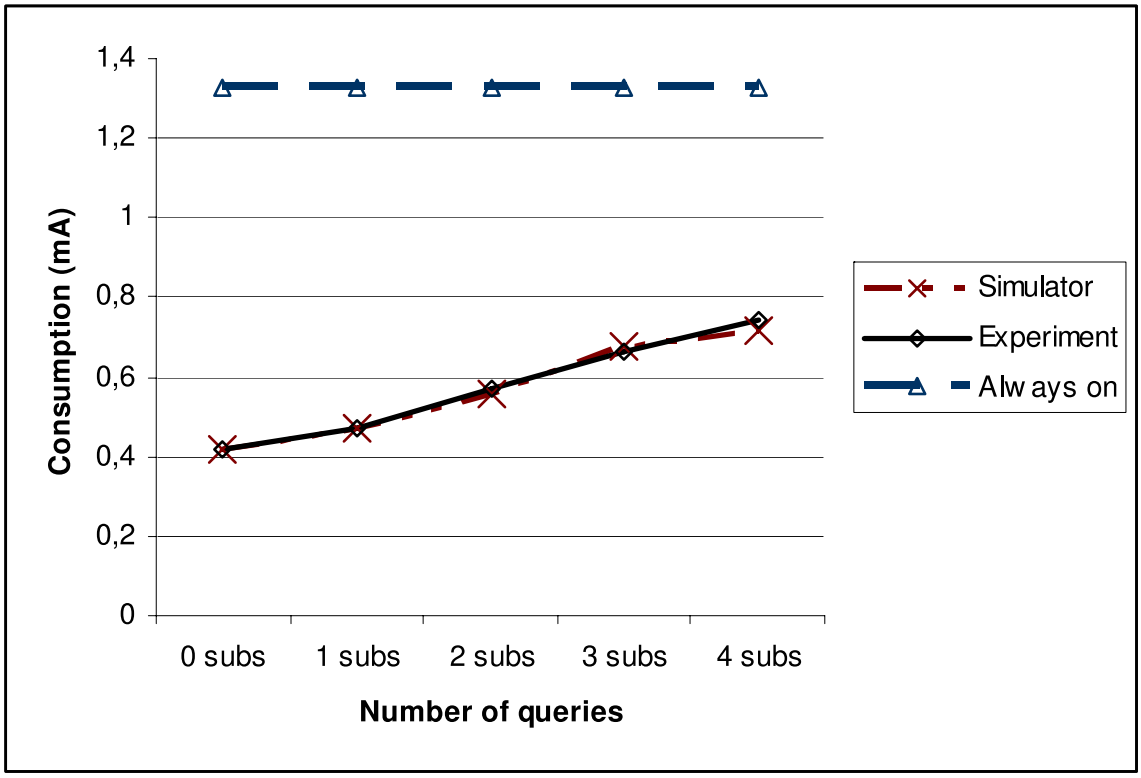


Figure 3.5: Performance of the MaD-WiSe energy efficiency mechanism.

in each experiment. Each experiment is repeated 10 times for 180 seconds, and in each experiment we measure the average period of time in which the radio of sensor s2 is in the following states: idle, ready, Rx and Tx. From these data we obtain the average energy consumption of sensor s2 (expressed in mAh) in all the sets of experiments, as shown in Figure 3.5. The figure also reports the energy consumption estimated with the TOSSIM simulator, and the energy consumption in the case where the energy efficiency mechanism is disabled. From the figure it is seen that the energy efficiency strategy enables significant energy savings, and that the energy consumed grows sub-linearly with the number of queries.

3.5 Execution of a query: a walk-through example

In this section we describe step by step all the phases executed by the system in executing the example query reported in Table 3.3. The query performs a timestamp join among values acquired by magnetic, acceleration, and temperature transducers on nodes 1, 2, and 3, respectively. When the condition defined in the **WHERE** clause is true, the result of the query is sent to the sink. Given that the asterisk

"$*$" is used in the **SELECT** clause, no projection is executed on the result, and all fields of the tuples that satisfy the condition are sent to the sink.

In order to be executed, the query has to go through 4 steps. 1) The query is parsed, and after syntax and type checking an initial query execution plan is generated. 2) The query execution plan is analyzed by the query optimizer that generates an alternative, semantically equivalent, query execution plan that requires lower energy consumption to be executed on the nodes of the WSN. 3) The optimized query execution plan, consisting of streams connecting operators, is injected in the WSN. 4) Each node of the WSN executes its portion of the query execution plan.

Steps 1) and 2) are executed at the sink (for instance a PC, a palmtop, etc). Step 3) is jointly executed by the sink and nodes of the WSN. Step 4) is exclusively executed by nodes of the WSN. These 4 steps are better described in the following.

1) Initial query plan generation: The query parser takes the query expressed using the MW-SQL syntax and translates it into a query execution plan consisting of operators of the query algebra connected by streams. The query plan generation starts from the **FROM** clause, where the source of the query is specified. The generated query execution plan can be seen in Figure 3.6a. According to the query in Table 3.3, the leaves of the tree representing the query execution plan are three periodic sensor streams representing magnetic, acceleration, and temperature transducers from nodes 1, 2, and 3, respectively, where the three streams are allocated. The **FROM** clause also specifies that a timestamp join should be executed among the tuples received by the three streams. Given that the timestamp join is a binary operator, the join among the three streams is obtained by using two cascading binary joins. In Figure 3.6a the two join operators are assigned to nodes 2 and 3. However, given that this is the initial query execution plan, this is rather arbitrary and the query optimizer will improve the allocation strategy. The query parser takes the selection operators to be added to the query execution plan in the **WHERE** clause. In Figure 3.6a the selection operators corresponding to predicates p_1, p_2, and p_3 are simply appended on top of the top-most join operator, and are assigned to node 3. As before, this is again arbitrary, and the optimizer will chose a better placement. Given that there is no projection defined in the **SELECT** clause, the query execution plan is almost complete. Connections between operators are realized with streams. Operators assigned to the same node are connected with local streams, while operators assigned to different nodes are connected with remote streams. Finally, the acquisition rate is taken from the **EVERY** clause and used to set the periodic sensor stream acquiring rate.

2) Query plan optimization: Once the initial query execution plan is generated it is given to the query optimizer to generate a better and equivalent one. Figures 3.6b and 3.6c show two examples of better, yet semantically equivalent, query plans. This is discussed in more detail in section 3.6.

3) Query plan dissemination: The optimized query plan is then disseminated in the WSN. To do so, messages corresponding to the operators to be executed and the streams to be set-up are sent to the nodes involved in the execution of the

SELECT *   **FROM** 1.Magnetism, 2.Acceleration, 3.Temperature   **WHERE** p_1(1.Magnetism) **and** p_2(2.Acceleration) **and** p_3(3.Temperature)   **EVERY** 3000

Table 3.3: Query used for the query execution and optimization example. The query performs a timestamp join between Magnetism, Acceleration, and Temperature readings from nodes 1, 2, and 3, respectively, every three seconds. If predicates p_1, p_2, and p_3 are satisfied, results are sent to the sink

query. For instance, if the optimized query execution plan is the one given in Figure 3.6c, node 1 receives requests to set up a sensor stream (associated to its magnetic transducer), a selection operator, and a remote stream towards node 2. The sensor stream is connected to the selection operator on the same node, which in turn is connected to the join operator, instantiated on node 2, by means of the remote stream. Similar requests are sent to the other nodes involved in the query.

4) Distributed query plan execution: When all relevant nodes have received their portion of query execution plan, a $query - start$ message is broadcasted and the query execution starts in all nodes in a distributed fashion. The query execution process is actually driven by periodic sensor streams. They acquire data at a fixed rate and pass the corresponding tuple (including the data and the timestamp) to the operator connected to them, which, after processing it, may pass the result to the next operator. For instance, in the query execution plan given in Figure 3.6c, the query execution is driven by the periodic sensor stream associated with the magnetic transducer in node 1. The acquired tuple is given to the selection operator on the same node. If the tuple satisfies predicate p_1, the tuple goes through a remote stream to the sync-join operator on node 2. The sync-join operator, as soon as it receives the tuple, asks the on-demand sensor stream on node 2 to acquire the acceleration. The tuple, obtained after the join, is given trough a local stream to the selection operator on node 2. If the tuple satisfies predicate p_2, it is sent to node 3, etc. This process is repeated every time the periodic sensor stream on node 1 acquires a new tuple.

The output of the query execution is a stream of the tuples resulting from the execution of the topmost operator on the query execution plan. These tuples are sent to the sink to be used by the application running on it.

3.6 Query optimization and results

While in traditional databases the Query Optimizer looks for a query plan such that the query execution is faster, in MaD-WiSe the focus of the optimization is the energy consumption. The Optimizer thus tries to find an optimal cost query plan,

where the cost is measured in terms of the energy required to compute a query.

MaD-WiSe uses an algebraic optimization approach, that is based on transformation rules transforming a query plan into a semantically equivalent one with a lower cost. The final query plan is obtained by applying successive transformations to an initial query plan built from the MW-SQL query. It also uses some strategies for re-ordering of the operators according to selectivity of predicates, cost of acquisition, and topology.

MaD-WiSe relies on some well-known transformation rules proposed in the literature to optimize traditional database query execution. For instance, rules to push-down selection and projection operators, and selectivity-based ordering of selections are very useful since they contribute to reduce the amount of data to be transferred upward in a query plan. This implicitly reduces the amount of data traversing remote streams, and, in turn, it reduces the amount of radio activity and of energy consumed.

In particular, MaD-WiSe uses the transformation rules defined according to the following guidelines:

SJ rule (Sync-Join): Sync-join and on-demand streams should be used whenever possible, to reduce cost of acquisition.

LDJT rule (Left Deep Join Trees): Given that a sync-join requires a sensor stream on the right side, trees representing query plans should be unbalanced to the left. In this way, the chance that a sensor stream (a leaf node) is found as the right argument of a join is increased.

PD rule (Push-Down): Unary operators such as selections, projections, and temporal aggregates (which reduce the amount of data being forwarded) should be moved as close as possible to the node where data is acquired, to reduce the cost of communication.

Given a query plan, the Optimizer executes three sequential steps:

1. Heuristically uses the transformation rules that apply to it, until there are no transformation rules that can be applied.

2. Performs operator re-ordering, according to the selectivity of predicates, cost of acquisition, and topology criteria.

3. Evaluates all the query plans obtained and, according to a proper cost model, it chooses the query plan that consumes less energy.

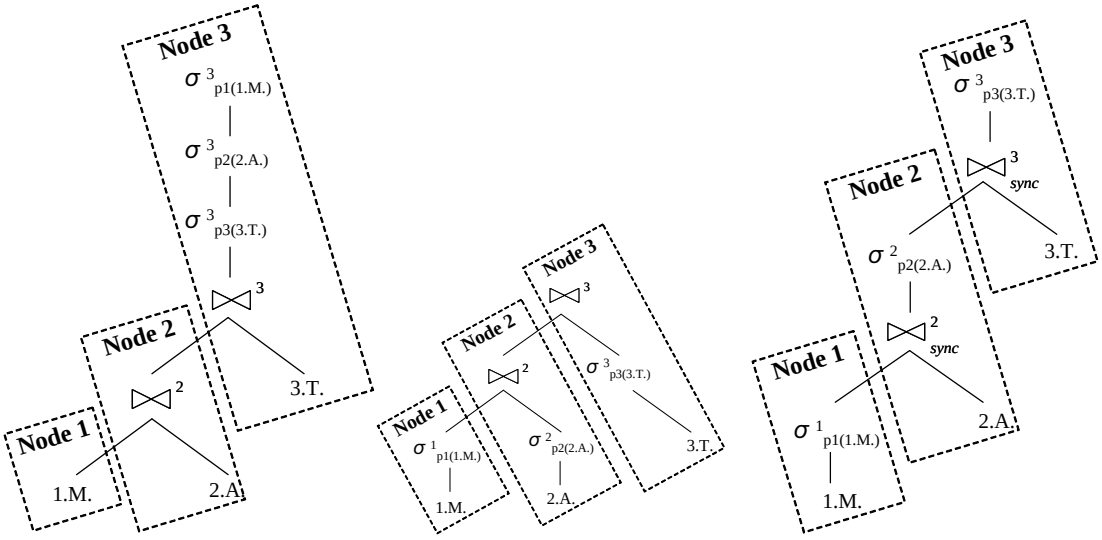
Optimization strategies, analysis and experiments

Let us assume that we submit the MW-SQL query showed in Table 3.3, where p_1, p_2, and p_3 are some predicates on magnetism, acceleration and temperature readings,

with probability $\texttt{Pr}(p_1) = 0.005, \texttt{Pr}(p_2) = 0.025, \texttt{Pr}(p_3) = 0.05$, respectively. These probabilities represent the selectivity of the predicates, and they are used by the query optimizer to perform the re-ordering of the operators.

Figure 3.6 shows three possible equivalent query plans that can be considered by the query optimizer to process the above query. The sink is assumed to be an external node connected to sensor 3. QP1, on the left, is obtained by applying the LDJT rule. It first acquires all specified data, and then joins them before applying the three selections on the last sensor. This requires all magnetism readings to be sent to sensor 2 and joined with the acceleration readings. The result of the join is sent to sensor 3 where it is joined with the temperature reading; then, the three selections are applied. QP2, in the middle, is obtained from QP1 by using the PD rule. In this query plan all data must be acquired. However, the magnetism reading is sent to sensor 2 only if it satisfies p_1. The join on sensor 2 is thus executed only if both p_1 and p_2 are satisfied, and in this case the result is sent to sensor 3. The join in sensor 3 is executed only if all the three predicates are true, and in this case the result is sent to the sink. QP3, on the right, is obtained from QP2 by using the SJ rule. In this case, magnetism is always acquired, and it is sent to sensor 2 if it satisfies predicate p_1. Only in this case the acceleration is acquired and joined with the magnetism value. The result of the join is sent to sensor 3 if p_2 is satisfied and, as a consequence, temperature is acquired. Finally, if p_3 is satisfied, the result is sent to the sink.

The costs of the three query plans are reported in Table 3.4. The costs reported in this table refer to the MicaZ motes, and they are obtained by analysis. In particular, for each query plan QP1, QP2, and QP3 the energy cost of each component of the sensors that is activated by the query plan, for a single round of data sampling, is considered. The energy cost of each component is taken from the MicaZ datasheets [5]. The transmission cost in the table includes both the send and receive costs for the sensors involved. In the table the total energy of each operation is computed by taking into account the probabilities $\texttt{Pr}(p_1), \texttt{Pr}(p_2), \texttt{Pr}(p_3)$ of success of the predicates p_1, p_2, and p_3, respectively. These probabilities affect the frequency (shown in column Freq.) with which the single operations involved into the query plans are executed. This is why, for example, the power required for the transmission of the magnetism in QP1 is always accounted for in the total energy cost of the query, while it is accounted with a weight equal to 0.005 in QP2 and QP3. The lower cost of QP2 with respect to QP1 is due to the reduced number of communications that it requires. The lower cost of QP3 with respect to QP2 is due to the combined reduction of communications and acquisitions. In this simple example, the improvement of QP3 with respect to QP2 is limited. However the next section shows how the use of sync-joins (as produced for QP3) with appropriate ordering of operators can provide significant performance improvements in more general cases.

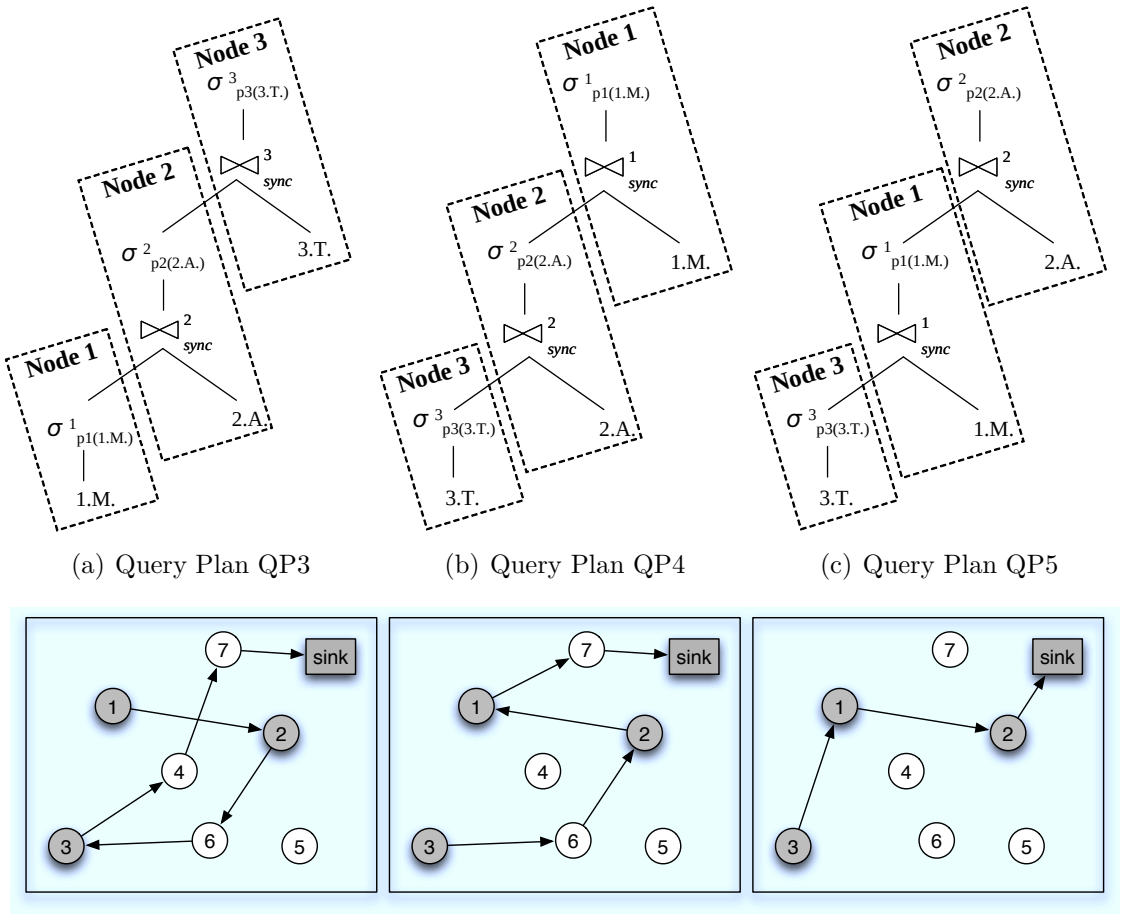


(a) Query Plan QP1 (b) Query Plan QP2 (c) Query Plan QP3

Figure 3.6: Three possible execution plans for the same query.

		QP1:		**QP2:**		**QP3:**	
Action	**Energy(mJ)**	**Freq.**	**Power**	**Freq.**	**Power**	**Freq.**	**Power**
Acquire M.	0.2685	1	0.2685	1	0.2685	1	0.2685
Send M.	0.31087	1	0.31087	0.005	0.00155	0.005	0.00155
Acquire A.	0.03222	1	0.03222	1	0.03222	0.005	0.00016
Send M.A.	0.31087	1	0.31087	0.000125	3.89E-5	0.000125	3.89E-5
Acquire T.	0.00009	1	0.00009	1	0.00009	0.000125	1.11E-08
Send M.A.T.	0.31087	6.25E-6	1.94E-06	6.25E-6	1.94E-06	6.25E-6	1.94E-06
Total Cost:			0.92254		0.3024		0.2702

Table 3.4: Costs of the three executions plans in Figure 3.6.



(a) Query Plan QP3 (b) Query Plan QP4 (c) Query Plan QP5

(d) Geographical placement of sensors and remote streams connecting the sensors. Grey nodes represent the sensors involved in the query, white nodes represent other sensors, that can be used as intermediate sensors that forward packets. Arrows represent remote streams used to send data among sensors.

Figure 3.7: Three possible execution plans for the same query using joins.

Ordering of operators

Several equivalent query plans that maintain the same overall structure can be obtained by changing the order of the operators in a tree. MaD-WiSe considers the following three different ordering criteria:

criterion S (Selectivity): more selective selections are pushed down in the tree;

criterion P (Power): less expensive transducers are pushed down in the tree;

criterion T (Topology): short range communications are given priority;

The first criterion (S selectivity) gives precedence to very selective predicates to filter immediately useless data, thus reducing communications and data acquisitions

QP3:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire M.	0.2685	1	0.2685
Transmit M.	0.31087	0.005	0.00155
Acquire A.	0.03222	0.005	0.00016
Transmit M., A.	0.62174	0.000125	7.77E-5
Acquire T.	0.00009	0.000125	1.11E-08
Transmit M., A., T.	1.24347	6.25E-6	7.77E-06
Total Cost:			0.2703

QP4:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire T.	0.00009	1	0.00009
Transmit T.	0.62174	0.05	0.03109
Acquire A.	0.03222	0.05	0.0016
Transmit T., A.	0.31087	0.00125	0.00039
Acquire M.	0.2685	0.00125	0.00034
Transmit T., A., M.	0.62174	6.25E-6	3.89E-06
Total Cost:			0.03351

QP5:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire T.	0.00009	1	0.00009
Transmit T.	0.31087	0.05	0.0155
Acquire M.	0.2685	0.05	0.01342
Transmit T., M.	0.31087	0.00025	7.77E-5
Acquire A.	0.03222	0.00025	8.05E-6
Transmit T., M., A.	0.31087	6.25E-6	1.94E-06
Total Cost:			0.02914

Table 3.5: Cost of the query plans QP3, QP4, and QP5.

by means of sync-joins. This criterion exploits (if available) the probability of success of the predicates involved in the query. The second criterion (P power) gives precedence to low cost acquisitions. High cost acquisitions are thus executed with low probability since they are high in the tree, and the data collected at the lower levels of the tree must pass the selections first. The third criterion (T Topology) reduces the communication costs by choosing an ordering of the operators and their allocation on the sensors such that the multi-hop communication paths are shortened.

Figure 3.7 shows the different query plans obtained by applying these criteria. Differently from the previous section, multi-hop paths are taken into account here. The figure also shows an hypothetical placement in the space of nodes involved in the query, and the remote streams that connect nodes to execute the query. Streams are labelled with the hypothetical number of hops needed to support communication through them. In QP3 operators are ordered according to criterion S; criterion P

	case 1 - p_1, is false	case 2 - only the lowest predicate is true	case 3 - only the upmost predicate is true	case 4 - p_1, p_2, and p_3 are all true
QP3	1.5393	1.8901	2.5274	3.4832
QP4	1.27089	1.9403	2.5274	3.1646
QP5	1.27089	1.8579	2.2088	2.5274

Table 3.6: Costs of QP3, QP4, and QP5 obtained from the experiments.

is used in QP4, and criterion T is used in QP5. Their costs are reported in Table 3.5. The costs in this table refer to MicaZ motes, and they are obtained by analysis. In particular, for each query plan QP3, QP4, and QP5, the energy cost of each component of the sensors that is activated by the query plan, for a single round of data sampling, is considered. The frequency at which the component is activated (reported in column Freq. in the table) depends on the probability of success of the predicates p_1, p_2, and p_3.

The cost of QP4 (0.033 mJ) is one order of magnitude smaller than the cost of QP3 (0.27 mJ). The cost of QP5 (0.029 mJ) is slightly smaller than the cost of QP4. Therefore, the expected lifetime of a network running QP4 or QP5 is about 8 times longer than when running QP3. However, this is not a proof that ordering according to the topology of the network is always the best solution. The results can vary depending on the selections filtering and on the acquisitions costs. In general, there is not an optimal ordering strategy. For this reason the Optimizer generates different orderings according to the various criteria, and chooses the one providing the best performance, according to the cost model adopted. As shown in the example, this may lead to performance improvements of orders of magnitude.

These analytical results have also been confirmed by experimentation where the three query plans QP3, QP4, and QP5 have been executed on a MicaZ sensor network. For each sensor have been measured the duration of the periods in which its radio is in idle, off, transmit, and receive mode, and the duration of the periods in which the transducers are active. Then this information has been crossed with the energy cost per unit of time taken from the datasheets of the MicaZ motes. The results of this experiment are reported in Table 3.6, that shows the cost of the three query plans for different values of the tree predicates p_1, p_2, and p_3. In particular, the table shows the results in four cases: case 1 - the lowest predicate is false (it is p_1 in QP3 and p_3 in QP4 and QP5), case 2 - only the lowest predicate in the query plan is true, case 3 - only the upmost predicate is false (it is p_3 in QP3, p_1 in QP4, and p_2 QP5), and case 4 - all the three predicates are true.

Scalability of the query

As the size of the WSN scales up, both the number of sensors involved in a query and the distance (in terms of number of hops) between them may also scale up. However, these facts do not affect the complexity of the tasks to be executed by

each individual sensor. In fact, MaD-WiSe constructs the query plan as a binary tree in which each sensor is a node of the tree and it aggregates and processes the input from its two children (if any). This means that, despite the number of sensors involved in the query and their distances, the amount of work assigned to each sensor in the query plan is limited, and the query optimizer enforces this property by imposing that the number of tasks assigned to a sensor in a query plan does not exceed its capacity. However, as the query gets more complex and the distances among the sensors involved in the query increase, the global cost of the query also increases, mainly due to the larger number of sensors involved either in the query or in the communications.

In order to evaluate this effect consider the query in Table 3.3 and its query plan QP5. This query, that involves three sensors and a sink, can be executed both in small and in large networks. Hence, it can be used to evaluate how it scales as the distances among the sensors involved in the query increase. In particular, the distance between sensors is expressed in hops, and the values analyzed are 1, 5, 10, 15, and 20 hops. The result of this analysis is shown in Figure 3.8 that reports, for each value of distance between the sensors, the cost of the query plan QP5 in the same cases considered in the previous section, i.e.: case 1 - the lowest predicate p_3 is false, case 2 - only the lowest predicate p_3 is true, case 3 - only the upmost predicate p_2 is false (i.e. p_3 and p_1 are both true), and case 4 - all the three predicates are true. The figure shows that the cost of the query plan scales linearly with the distances in all the four cases, and as the distances increase the cost of transmission becomes dominant. The differences between the four curves are due to the fact that, when some of the predicates are false, the sensors avoid the cost of transmitting and receiving packets. For example, if p_3 is false (case 1) then no packet is actually sent, and thus the total cost accounts for merely the cost of keeping the radio of the sensors in idle mode, while in case 4 all the predicates are true; hence, the total cost accounts for both the idle mode of the radio and the transmission and reception of the packets.

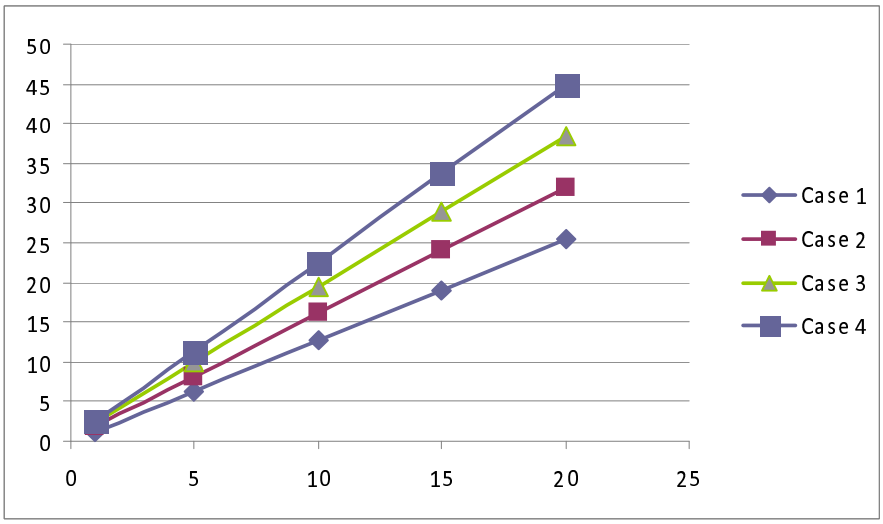


Figure 3.8: Cost of the query plan QP5 as the distance in hop between the sensors increases.

Chapter 4

Detection and Tracking of Mobile Events

Most of the current query processing approaches are inefficient when tracking mobile events, since they require continuous updates to the query every time the event moves or changes its shape and size. Furthermore, the queries expressed with these approaches usually address the individual transducers, while in tracking an event a user is mostly interested in information about the event such as speed, direction, size and shape, information that cannot be obtained by reading a transducer.

In this section we present a proposal for the detection of moving events and for their tracking. Specifically, we model the concept of composite event in WSN in Section 4.1, and we define the Event Query language (EQL) for the detection and tracking of such events in Section 4.2. EQL is a declarative query language that allows the definition of composite events, and also of detection and tracking tasks. In Section 4.3 we propose a query processing mechanism for the efficient execution of EQL queries, and we model this mechanism by means of finite state machines in Section 4.4. In Section 4.5 we evaluate the proposed approach, showing that it has a lower overhead in terms of energy consumption, and that it scales better with the mobility of the tracked events when compared to a centralized query approach.

4.1 Modeling Events as Query-able 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 4.1.

The occurrence of an event can also produce the occurrence of other dependant events. For instance, we can suppose that, after the explosion, it is possible that a

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

Table 4.1: 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 4.2: The query requests the position and the event speed GasCloud.

"gas cloud leak" event may also occur. As before, the event is detected by checking the occurrence of a combination of measured values. However, given that the event depends on another event, the condition should be checked just after the occurrence of the 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 once they first occur. The gas cloud can move in the environment, can expand, or change density, etc. Therefore, in many cases, in addition to the fact that an event has occurred, it might also be useful to track the event in space and to monitor its evolution.

Once an event has been detected and is being tracked, we might be interested in obtaining information about the event itself. For instance, once we detect that a gas cloud has been produced we might 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 directly use events as data sources. A very simple example of a possible query on the gas cloud event is given in Table 4.2.

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 4.2 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 just has 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 when the event is first defined, and are hiddenly 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 allow to:

1. define events and data that can be read by events;
2. define and process event detection tasks;
3. define and process tracking tasks;

4. express and process queries that have events as data sources.

In the rest of the paper, Section 4.2 will address these four points defining EQL (Event Query Language), a query language for querying and tracking events; Section 4.3 proposes an in-network query processing solution for this language.

4.2 Declarative Language for Event Detection, Tracking, and Querying

The language that we propose in this section allows the definition of:

1. events in terms of conditions on values measured by transducers in the environment, and values returned by the event once detected;
2. rules for detecting the event;
3. rules for tracking the event during its evolution;
4. 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 more detail.

4.2.1 Event statement

In order to define a new event we need to specify the condition 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 size 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 4.1). The decision about the occurrence of an event is taken by using the information acquired by all sensors in the radius defined by the

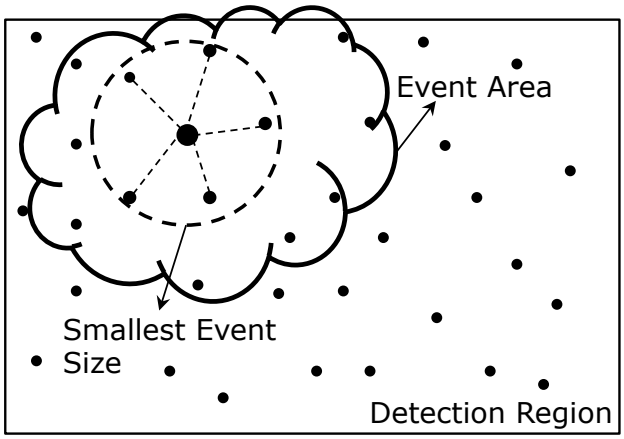


Figure 4.1: Detection Region - In the figure the external rectangle represents the Detection Region where the event is monitored, the dotted circle represents the minimum expected size of the event (the Smallest Event Size that in this case is 1 hop) related to the sensor represented with the big black spot, and the cloud represents the actual area covered by the event when it occurs (the Event Area).

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 4.3 reports the syntax of the Event Statement. We explain this syntax, and that of the other statements, building an example that defines explosion and the gas cloud events. Tables 4.4 and 4.5 contain the definitions of these two events.

The clause `SIZE` specifies the extent of the Detecting Area 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 of at least n, contribute to get a value above the threshold.

For instance, in the examples in tables 4.4 and 4.5, the clause `SIZE` specifies that the Smallest Event Size is 2 and 3 hops, respectively (if sensors are aware of their position, size can be expressed in terms of 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 clause `AS` 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 will be obtained as aggregates. For example, in Table 4.4, the values returned by the explosion event are the noise, the average acceleration, and the average pressure computed in an area of size 2 hops.

In Table 4.5 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 occurred. Values defined in the `AS` clause can be used here.

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

Table 4.3: 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 4.4: 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 4.5: The GasCloud definition.

4.2.2 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, detection of an event is not necessary to be monitored in the whole environment covered by the WSN. Rather, just nodes deployed on some specific critical regions have to execute the detection task. After an event occurs, if the event moves, neighbor nodes will be alerted as well. We call the critical region of the environment, where an event is initially monitored, the *Detection Region*. For

instance, only the nodes close to the gas tank need to execute the detection task for 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 frequency at which the event conditions should be checked, and other events that the current event might depend on.

The Detection Region identifies the set of nodes of the WSN where the occurrence of the event will be initially monitored. All nodes in this area are instructed to perform the detection task [56]. When an event occurs, a subset of the nodes of the Detection Region within an area of size Smallest Event Area (as specified in the Event Statement), will cooperate to detect the event (see Figure 4.1).

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 they manifests themselves very fast.

The syntax of the Detection Statement is shown in Table 4.6.

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

Table 4.6: The Detection Statement.

The clause `ON REGION` 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 sensor identifiers that are known to belong 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 contributed to detect the named event. This area is larger than the Detection Area of the named event (see Figure 4.1). 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 4.7: The detection definition for the Explosion event.

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

Table 4.8: The detection definition for the GasCloud event that depends on the Explosion event.

Tables 4.7 and 4.8 show an example of the detection for the *Explosion* event and for the *GasCloud* event. In the example, the GasCould event depends on the explosion event. This means that the detection task for the GasCould event will start after an explosion, using the area where the explosion occurred as the Detection Region.

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

4.2.3 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. Tracking has the purpose of monitoring 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 atmospheric conditions, and it may expand or change its shape and size).

The Tracking Statement specifies how the event should be tracked, and also 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 will discuss in Section 4.3, 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 4.9 reports the syntax of the Tracking Statement, and Table 4.10 reports the Tracking Statement for the GasCloud event.

The clause `EVOLUTION` 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 4.10, it is expressed in number of hops.

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

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

Table 4.9: The Tracking Statement

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

Table 4.10: The Tracking Statement for the GasCloud

4.2.4 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 4.11: The Query Statement

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

Table 4.12: The Query statement for the Gas Cloud example.

Table 4.11 reports the syntax of the Event Statement, and Table 4.12 reports the Data Statement for the gas cloud example.

The clause `SELECT` 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 clause `FROM` specifies the event to be used as source of information. Of course, data will start coming from an event just after it occurs. The clause `WHERE` specifies a list of conditions that have to be satisfied on the properties of the event.

Table 4.12 contains a query on the GasCloud event. The example consists of a request for the position and the speed of the gas cloud only when the average value of the oxygen is below 50.

4.3 In-network Event Query Processing

In this section we describe in more detail the three main phases for processing EQL: detection task, tracking task and query execution task.

4.3.1 Detection Task

When a Query Statement is submitted, it is necessary to ask the nodes included in the Detection Region, of the event being queried, to start executing the detection task, unless they are already executing it. Clearly, a Query Statement cannot be submitted if the corresponding event has not yet been defined.

The pseudo-code of the detection task executed by all nodes of the Detection Region is reported in Algorithm 1.

The detection task, executed by a node, (1) acquires the needed data (as defined in the Event Statement) from local transducers and (2) broadcasts them to neighbors reachable up to S hops. Then, it (3) receives data acquired by its neighbors and (4) evaluates the aggregate operators and the condition to check if the event occurred. If the event did not occur, then it (5) repeats the previous steps. Elsewhere, (6) the set of sensors, which contributed to detect the event, are collected by means of the EventAreaDefinition procedure. After this, it launches the (7) tracking task, and it checks (8) if there is some other event depending from the detected one; in this case, the detection task of the next dependent event 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. This procedure is executed locally by all nodes involved in an event and the aim, shortly, is to provide all of them with the list of all nodes that detected the event. In order to prevent the procedure from running for a long time, (1) a timer is set to stop the execution. The list of nodes is (2) initialized with the node executing the procedure. This information is (3) broadcasted to the neighbors reachable up to S hops. Then, it (4) receives the list of nodes collected by its neighbors. The list of nodes collected is (5) updated by making the union of the previous list and the list received by the neighbors. If the new (6) list is different from the old one (it was actually updated), then it goes to (3). Elsewhere, (7) if the timer has not expired, it goes to (4). Eventually, (8) the list of the collected nodes is returned. Note that, with this procedure, all nodes have the full list of nodes that participated in the detection of the event, so all nodes become aware of the Event Area.

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 !expired(Timer) then
9:    go to 4
10: else
11:   return activeSensorList
12: end if
```

4.3.2 Tracking Task

Once the tracking task is triggered by the detection task, nodes involved in the event start executing it cooperatively, so that they can contribute to track and collect data from the detected event. The pseudo-code of the tracking task is sketched in Algorithm 3.

Nodes executing the tracking task (1) elect their leader, then they (2) build a routing and data collection tree. Nodes that are at the border of the event (that is, leaf nodes of the tree) alert (3,4) their neighbors, reachable up to S hops, to check if they also detect the event. Note that these nodes might currently be outside 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 (5) no longer see the event, and if

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 !checkInclusion(eventName) and expired(timeout) then
 7:    return
 8: else
 9:    activeSensorList=updateActiveSensorList(leader, tree)
10:    go to 1
11: end if
```

the timer has expired, they quit the Tracking task. Elsewhere, they (6) update the list of nodes involved in the event and (7) restart the tracking loop. 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, and that 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 the literature [42], [63] (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 on the tree with a counter equal to half 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 occurrences of the same kind of event occur in two disjoint 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.

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 commonly used in the literature [39], [40].

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 nodes in the boundary notify the alert to nodes reachable within the number of hops specified by the clause `EVOLUTION` 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 in the data collection tree, and they are notified about the leader (see Figure 4.2). Passive boundary nodes then start acquiring the data needed to detect the event according to the sampling period specified by the clause `EVERY`, and for a total time specified by the clause `TIMEOUT` 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, it executes the tracking task, and it 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 clauses `AS` and `WHERE` 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 a 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 4.2 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.

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 4.13: Glossary of the definitions related to the events.

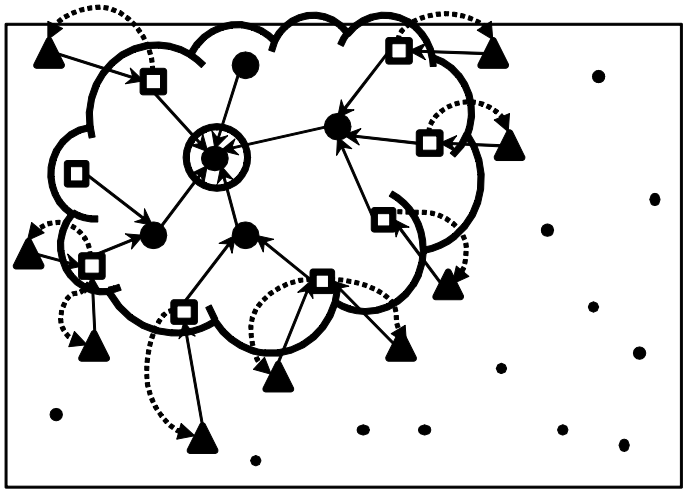


Figure 4.2: TRACKING PHASE - The figure shows the data collection tree built inside the Event Area. The tree is rooted in the leader sensor, represented as the rounded circle. The figure also shows the updating operation: the active boundary sensors (the empty squares) alert the passive boundary sensors (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.

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 4.13 summarizes the definitions used in this Chapter.

4.3.3 Query Execution

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, and the tracking task builds and maintains the data collection tree involving the active sensors. The query execution task is executed by all the active sensors, and it uses the data collection tree to send to the leader the data about the event as specified in the Query Statement.

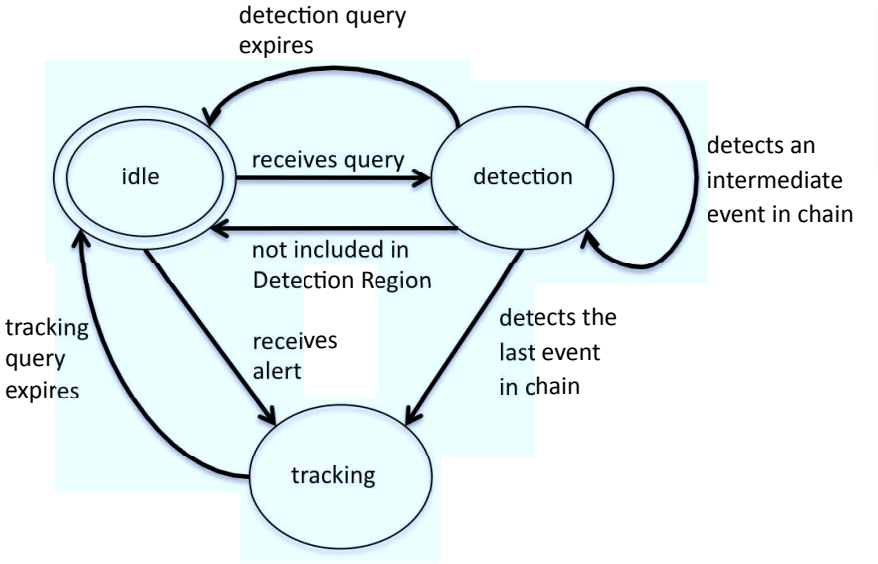


Figure 4.3: AUTOMATON - Abstract representation of the Finite State Automaton for EQL processing. The detection and tracking states are complex super-states that are composed of several internal states.

To this purpose, the root of the data collection tree, created by the detection task, and continuously updated by the tracking task, is used as a data source. Data arriving at the root of the tree are processed by using techniques for in-network distributed WSN query processing already available in the literature. These issues are, for instance, extensively addressed in systems such as MaD-WiSe [13] or TinyDB [69].

These approaches exploit classical relational operators [22] and data base query processing techniques, appropriately adapted to the WSN data management context.

4.4 Finite State Machine

Figure 4.3 represents a high-level abstraction of the Finite State Machine about the processing of an EQL query. Each sensor passes through three states, *idle*, *detection* and *tracking*, which are actually super-states that can be refined by several internal states. Note that a sensor can be involved in more than one query, so it can be executing more than one tracking task. 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**. 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

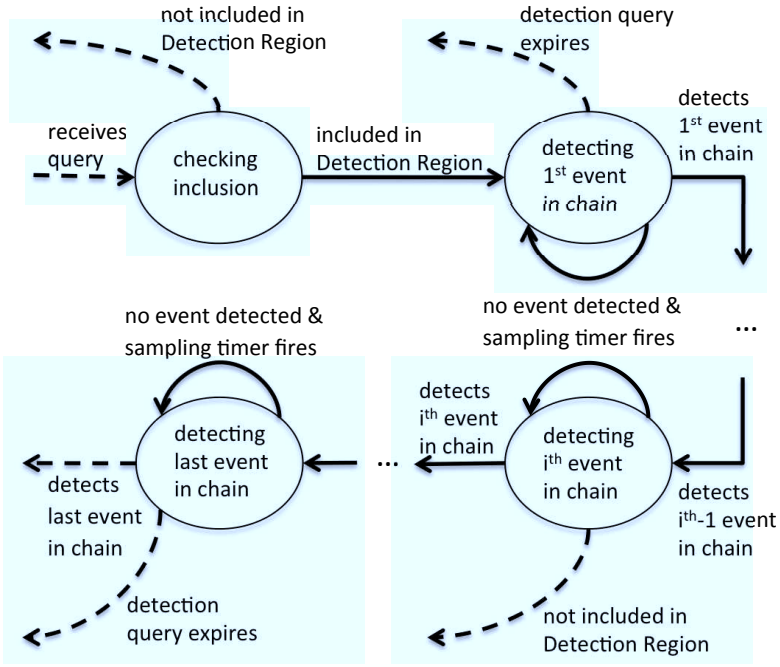


Figure 4.4: Detection State Refinement - The Figure reports the refinement to the detection super-state of the FSA reported in Figure 4.3. The dotted lines represent the input/output transitions to/from the detection super-state.

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 switches to the *tracking* state because it belongs to the Event Area of the event to be tracked (see Figure 4.4). 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 a 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 it 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.

In the following we provide a refined version of the *detection* and the *tracking*

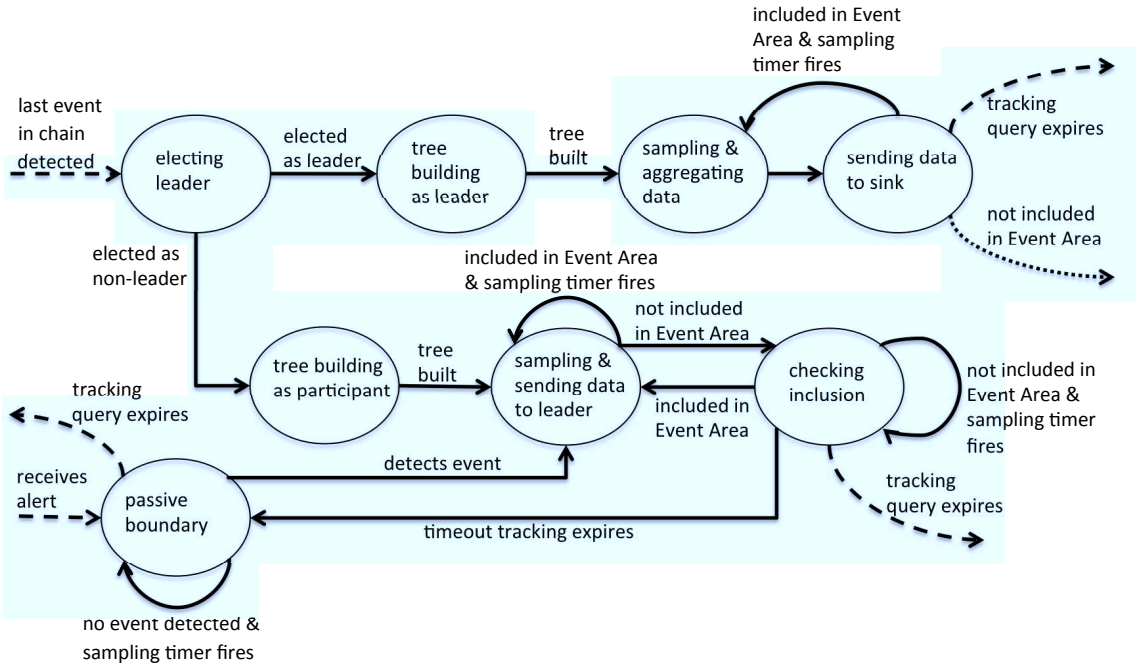


Figure 4.5: Tracking State Refinement - The dotted lines represent the input/output transitions to/from the tracking super-state. The dotted outgoing line related to the "sending data to sink" state is

states.

4.4.1 Detection State Refinement

A first refinement of the *detection* super-state is presented in Figure 4.4. 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 others 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.

4.4.2 Tracking State Refinement

The refinement for the *tracking* super-state is reported in Figure 4.5. 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, 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 state if the tracking timeout expires, or if the event moves and it is not included in the Event Area anymore. For the sake of simplicity 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 the branch executed by the leader. These sensors first participate to the tree building phase. Then they execute a loop in which they acquire the 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, in the latter it goes to the *checking inclusion* state. In this state it continues detecting 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.

4.5 Evaluation of EQL

In this section we present the evaluation of the proposed system in terms of power consumption and of 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 approach the sensors acquire the data and send them to a base station 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 base station can deduce the direction and the speed of the event, and thus it submits a new query to the sensors displaced in the area that will be covered by the event every time it moves for a distance equal 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 will present the MATLAB simulator that we used to perform the experiments. In particular, the simulation models generic

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

Table 4.14: Energy required for a sample from various transducers of MTS310CA boards.

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

Table 4.15: Energy required for sending and receiving a message of 50 bytes on the IRIS mote.

events with a circular shape, and assumes uniform rectilinear motion of events. 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 4.3.

4.5.1 Cost Model

The cost model takes into account the number of messages transmitted and received by the sensors, as well as 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 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 motes with MTS310CA sensor boards for the transducers [5] are used to implement the query, and we evaluate the costs referred to this hardware platform. Tables 4.14 and 4.15 report the related costs.

4.5.2 Network, Event and Simulation Model

We consider a network composed of sensors uniformly distributed over a square area of size $L \times L$ meters (see Figure 4.6), 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. The network density ρ (defined as the number of sensors per unit area) is given by $\rho = n_x/(\pi r_x^2)$. We assume that the

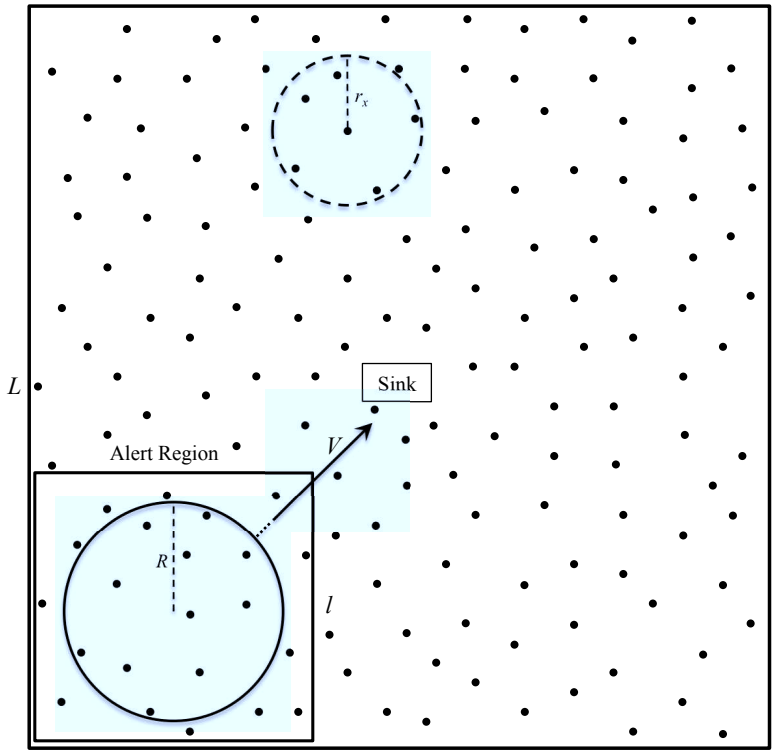


Figure 4.6: NETWORK - Each sensor has a transmission range r_x. The small square represents the Detection Region. The circle represents the Event Area, and it moves according to the rectilinear motion vector V.The sink is assumed to be in the center of the network.

Detection Region is a square area with side l placed in the bottom-left corner of the network, and that the Smallest Event Size is 1 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 topologies.

Table 4.16 reports the list of parameters that we kept fixed in our simulations, and Table 4.17 shows the parameters that we changed during the simulations.

4.5.3 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 ev1, with EQL.

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 4.16: 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 4.17: Variable parameters used for the analysis. In each scenario we study the behavior when changing one of these parameters and keeping the others fixed.

To this purpose, we analyze all phases executed during the detection and tracking (we assume that the query has already been injected), that are:

- **Detection** - detection of the next event in the chain of events.
- **Alert** - alerting of the neighbor sensors to be ready to detect the event.
- **Check Event** - detection, by the alerted sensors, of the currently tracked event in order to follow the evolution of the event.
- **Tree Updating** - updating of the Event Area and the data collection tree.

Note that the last three phases are executed periodically, according to the sampling period Δt.

Detection

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 1 hop). Then each sensor computes the average value of the received data and determines if the event occurred. Let A_r be the set of of sensors in the Detection

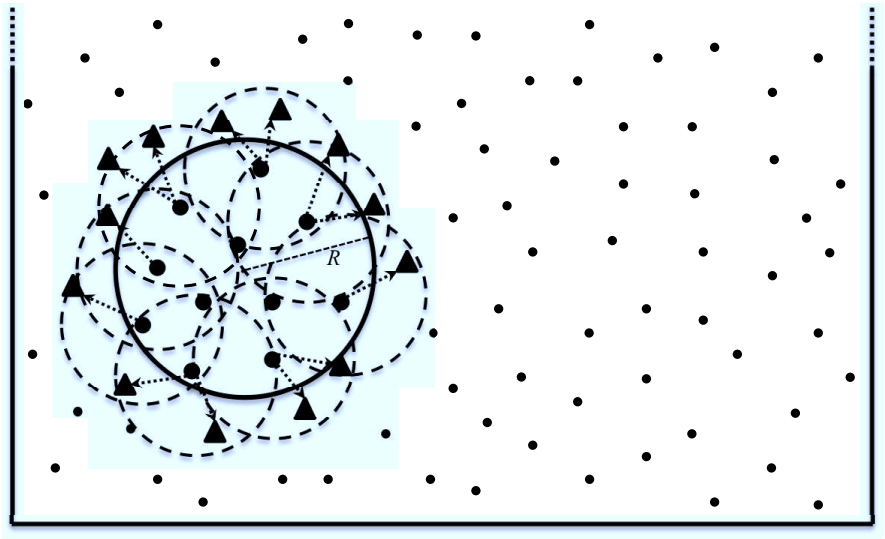


Figure 4.7: First Alert - The solid circle is the Event Area with radius R. The dotted circles represent the transmission of the alert messages and the triangles are the alerted sensors.

Region; the cost of one iteration of the detection is given by the cost of transducers activation times the senors in A_r plus the cost of a broadcast communication times the sensors in A_r:

$$c_{dt}^{EQL} = (\gamma + \delta)|A_r| \tag{4.1}$$

where $|A_r|$ is the cardinality of A_r. This is repeated, according to the specified sampling rate, until the event occurs.

Alert

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 big as specified in the clause `EVOLUTION` of the Tracking Statement.

Except for the first iteration, where all the active boundary sensors execute the alert operator (see Figure 4.7), at each iteration only the new active sensors (i.e. the sensors that become part of the Event Area at the previous iteration) alert their neighbors (see Figure 4.8). 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 is given by the cost of a broadcast communication times the sensors in D_s:

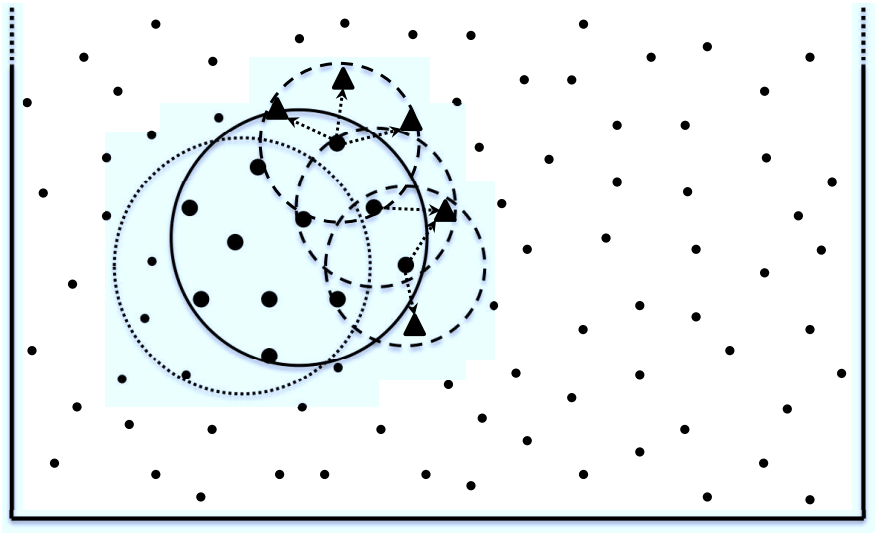


Figure 4.8: ALERT - The added sensors send the alert message.

$$c_{al}^{EQL} = \delta|D_s| \tag{4.2}$$

Check Event

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 is given by the cost of transducers activation times the number of sensors in the two sets C_s and L_s:

$$c_{ck}^{EQL} = \gamma|C_s \cup L_s| \tag{4.3}$$

Since the data to be returned to the user (i.e. the data specified by the clause `SELECT` 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

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

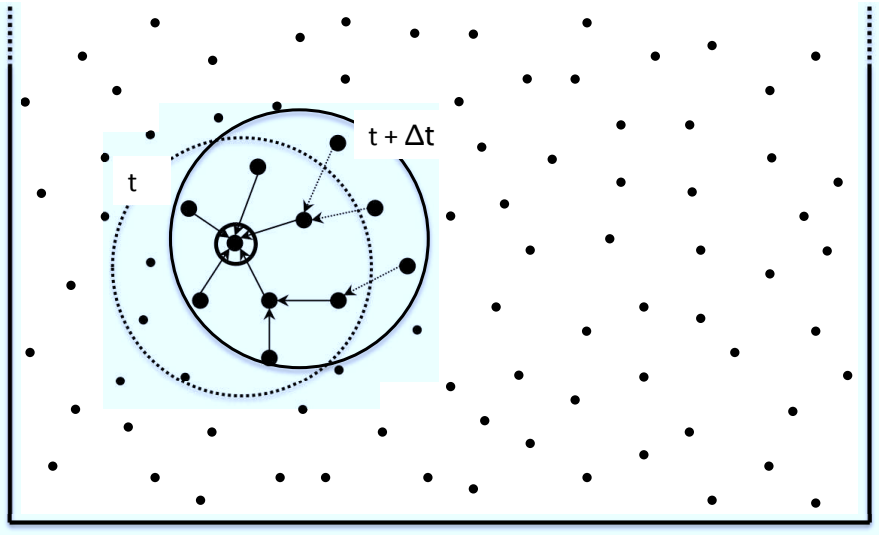


Figure 4.9: TREE UPDATING - The solid circle represents the Event Area at time $t + \Delta t$, while the dotted circle represents the Event Area at time t. The newly involved sensors in the lune are added to the data collection tree (the dotted arrows).

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 4.9). 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 an acknowledgement of the actual event detection. Thus, the cost of the Tree Updating phase is given by the cost of a unicast communication times the number of sensors in D_s:

$$c_{tu}^{EQL} = \beta |D_s| \tag{4.4}$$

4.5.4 Cost of CQA

As said in Section 4.5, 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 base station that performs the actual detection and pilots the tracking of the event. The 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 the following:

- **Detection** - detection of the next event in the chain of events.

- **Query Submission** - submission of the query to the center of the new Event Area.
- **Query Broadcast** - broadcast of the query inside the Event Area and building of the data collection tree.
- **Data Acquisition** - acquisition of the requested data.

Detection

This phase is executed in the same manner as EQL, and it has the same cost.

Query Submission

In CQA the sink has to renew the query when the event moves outside 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. For 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 for a distance equal to twice the radius of the event ($s = 2r$). Note that this is the minimum rate at which the sink can submit a new query without losing 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 euclidean 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 a path of length d_x. Therefore, the cost of the query submission in CQA is given by the number of the sensors thus computed times the cost of a unicast communication:

$$c_{qs}^{TDB} = \beta \lceil d_x / r_x \rceil \tag{4.5}$$

Query Broadcast

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, and it has a cost equal to the cost of a broadcast communication times the number of sensors in F_s:

$$c_{qb}^{TDB} = \delta |F_s| \tag{4.6}$$

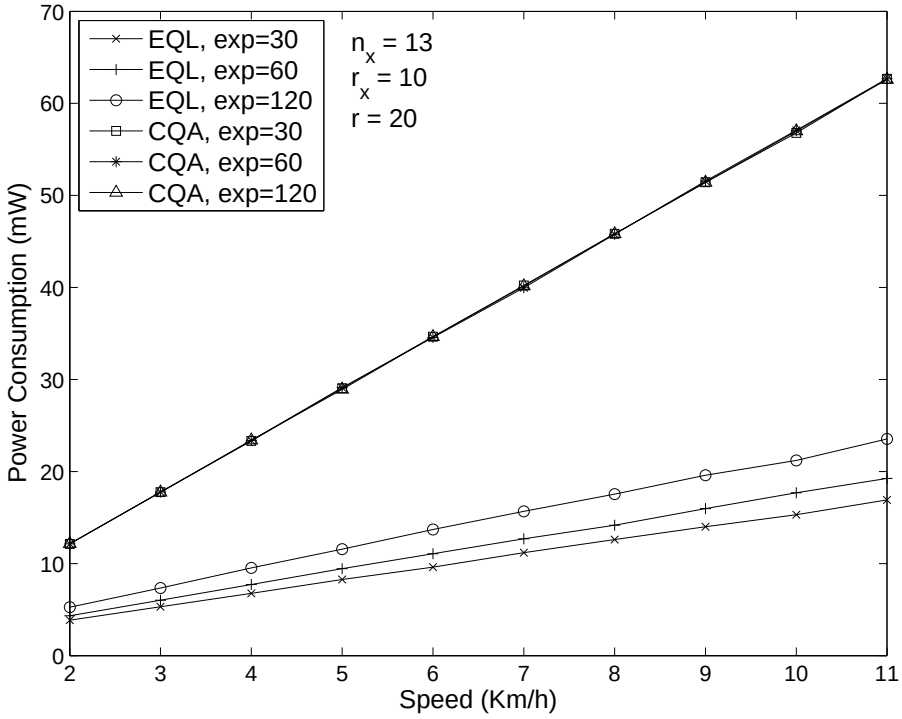


Figure 4.10: Power consumption with increasing values of event speed and different values of expiration time.

Data Acquisition

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 is given by the cost of the transducers activation times the number of sensors in F_s:

$$c_{da}^{TDB} = \gamma |F_s| \tag{4.7}$$

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 for a distance $s = 2r$.

4.5.5 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 4.17. 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

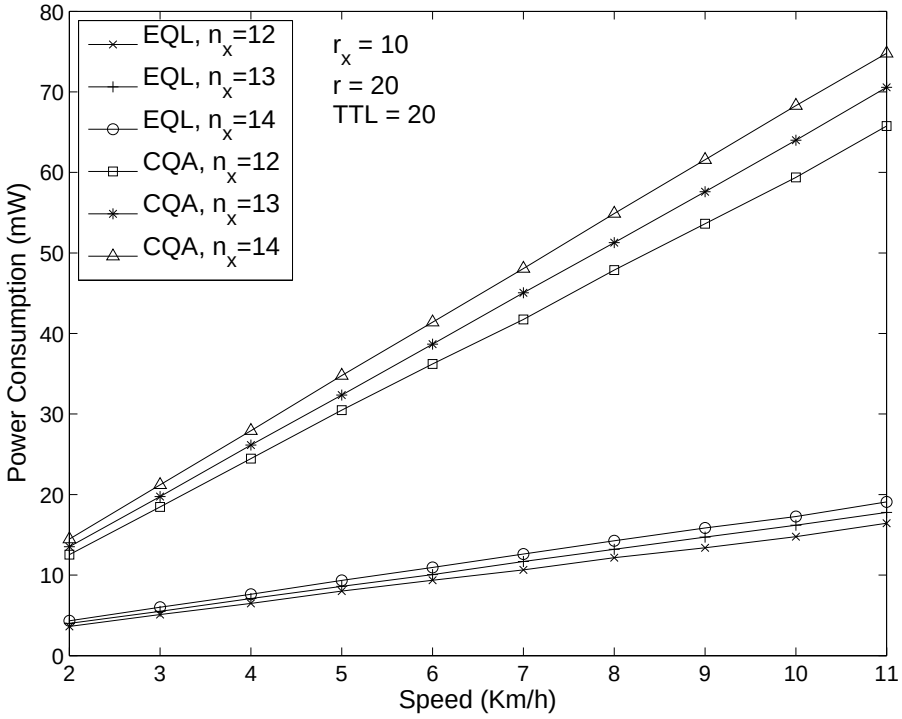


Figure 4.11: Power consumption with increasing values of event speed and different values of network density.

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 change 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 Figures 4.10, 4.11, 4.12 we report the power consumption of the two systems by combining increasing values of event speed and different values of expiration time, network density and size of the Event Area, respectively. 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, become involved in the event. This results in more communications and in more transducer activations. In all these experiments EQL shows a power consumption lower than CQA, and it also scales better as the studied parameters increase. This occurs because, each time that in CQA a new query is submitted, more sensors are involved in the query, and so the power consumption results higher. In EQL, on the other hand, the operations executed to track the event are local to the nodes in the boundary of the Event Area, and thus the power consumption in this case is less affected than in case of

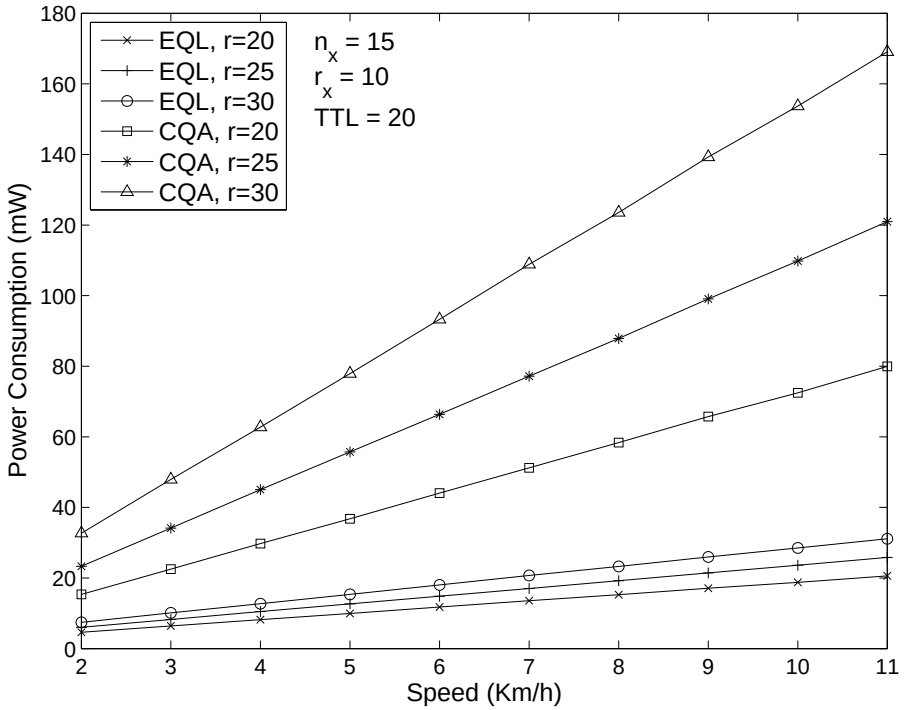


Figure 4.12: Power consumption with increasing value of event speed and different size of Event Area.

CQA.

In the scenario reported in Figure 4.10 we note that CQA is independent from the expiration time, since in this case the sensors are not alerted, while in EQL a higher value of this parameter results in a higher power consumption, since the sensors remain active in the monitoring phase for a longer time. More specifically, the power consumption of EQL, with expiration time of 30 seconds, ranges from 3,8 mW, with a event speed of 2 Km/h, to 16,9 mW if the event speed is 11 Km/h. With an expiration time of 60 seconds, the power consumption of EQL ranges from 4,3 mW, with an event speed of 2 Km/h, to 19,2 mW if the event moves at 11 Km/h. With an expiration time of 120 seconds, EQL has a power consumption that ranges from 5,2 mW, if the event speed is 2 Km/h, to 23,5 mW with an event speed of 11 Km/h. CQA, on the other hand, has a much higher power consumption as the event speed increases. In fact, its power consumption is 12,1 mW with an event speed of 2Km/h, while it is 62,6 mW if the event moves at 11 Km/h.

Figure 4.11 reports the power consumptions of the two systems in the scenario in which the event speed is studied with different values of network density. A higher number of neighbors per node results in a greater number of sensors involved in the event in each iteration, and thus in a higher power consumption. Also in this case, the power consumption of CQA is more affected than that of EQL because of the greater number of sensors involved in the event. In particular, with a number

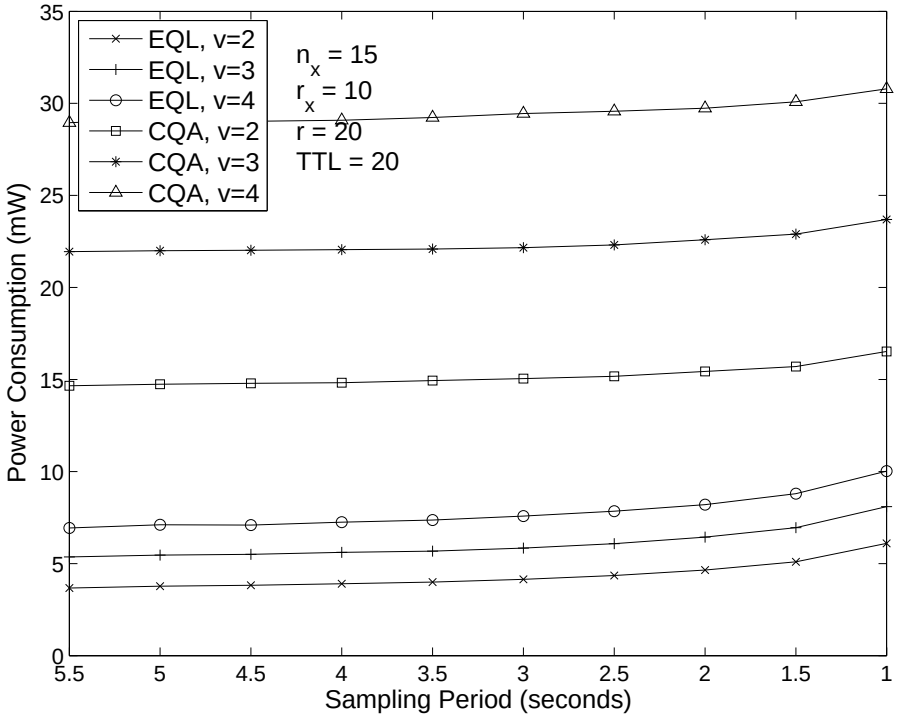


Figure 4.13: Simulation with increasing values of sampling rate of the sensors and different values of event speed.

of neighbors per node of 12, the power consumption of EQL ranges from 3,6 mW to 16,4 mW as the event speed grows from 2 Km/h to 11 Km/h, while the power consumption of CQA, with the same network density and the same event speed, ranges from 12,5 mW to 65,7 mW. With a number of neighbors per node of 13, EQL has a power consumption that ranges from 4 mW, with an event speed of 2 Km/h, to 17,8 mW, with an event speed of 11 Km/h, while CQA has a power consumption that ranges from 13,2 mW to 70,6 mW with the same values of network density and event speed. With a number of neighbors per node of 14, the power consumption of EQL ranges from 4,3 mW, if the event moves at 2 Km/h, to 19 mW, if the event moves at 11 Km/h, while the power consumption of CQA ranges from 14,5 mW to 74,8 mW, with the same values of network density and event speed.

In Figure 4.12 we can see the power consumption of EQL and CQA in the scenario in which the speed and the size of the event are studied. Also in this case, as for the network density, a bigger event has a larger number of sensors involved in it, and this results in a greater power consumption. CQA shows a higher power consumption with respect to EQL, and it 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

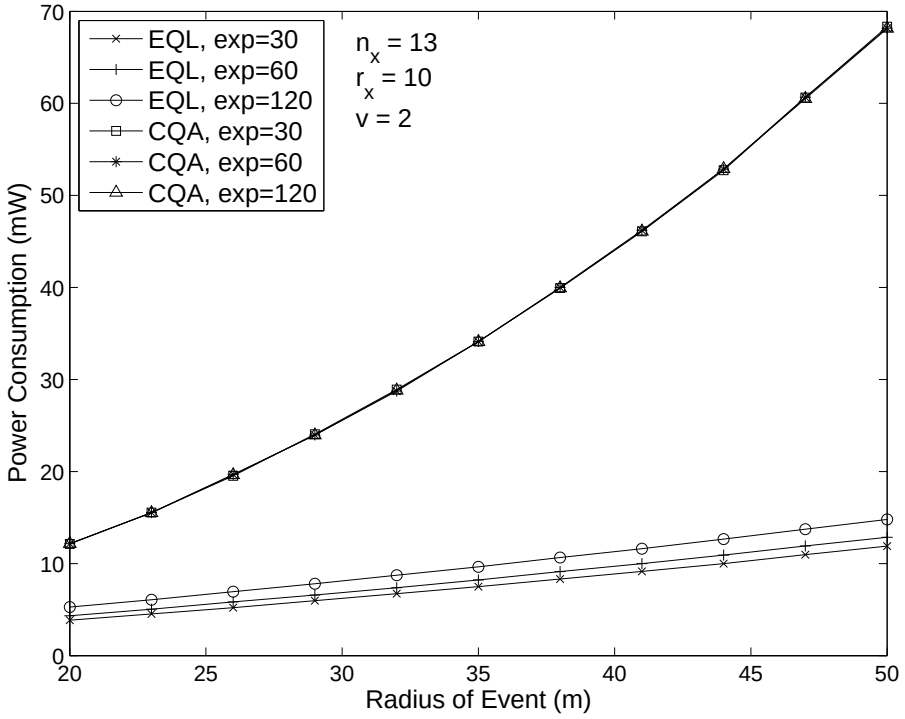


Figure 4.14: Power consumption with increasing size of the Event Area and different values of expiration time.

does not imply too much overhead in this case. In particular, 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. With an event radius of 25 meters, EQL has a power consumption that ranges from 6 mW, if the event moves at 2 Km/h, to 25,8 mW, if the event moves at 11 Km/h, while, with the same event conditions, CQA has a power consumption that ranges from 23,3 mW to 120,9 mW. With an event of radius 30 meters, the power consumption of EQL is 7,4 mW, with an event speed of 2 Km/h, and it is 31,1 mW, with an event speed of 11 Km/h. The power consumption of CQA with the same event speed and size, ranges from 32,7 mW to 169,1 mW.

Figure 4.13 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 in seconds 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 samples, and thus more communications overhead

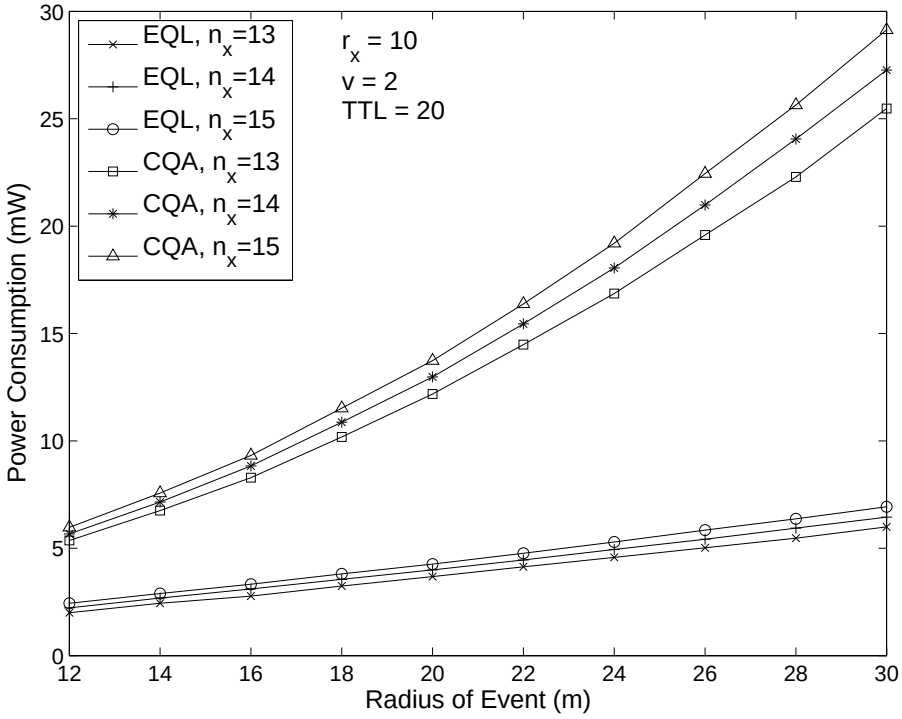


Figure 4.15: Power consumption with increasing size of the Event Area and different values of network density.

is necessary. 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 represents 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 sampling rate of 5,5 seconds, to 10 mW, with a sampling rate of 1 second, while the power consumption

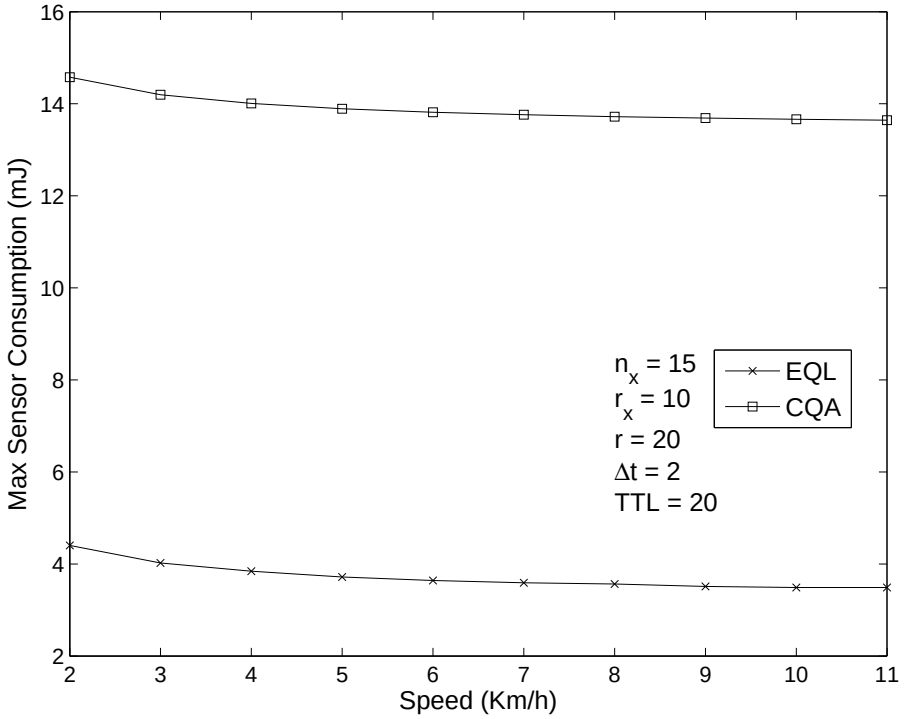


Figure 4.16: Maximum energy consumption of the sensors with increasing values of event speed.

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 Figures 4.14, 4.15 we analyze the power consumption of the two systems for increasing values of the size of the Event Area and different values of the expiration time and the network density, respectively. We can see that, in both cases, the power consumption increases with the size of the event, but this affects CQA much more than EQL, since a wider event involves more sensors that have to receive the query, so more communications are needed; in EQL the tracking of the event is executed locally by the sensors at the border of the event, requiring a much smaller overhead. Figure 4.14 shows also that CQA is independent from the expiration time parameter, since the sensors are not alerted. On the other hand, this affects the power consumption of EQL because, for a greater value of the expiration time, the alerted sensors have to monitor the event for a longer time. In particular, with an expiration time of 30 seconds, EQL has a power consumption of 3,8 mW, if the radius of the event is 20 meters, and of 11,9 mW if the radius of the event is 50 meters. With an expiration time of 60 seconds, the power consumption of EQL ranges from 4,3 mW, if the event has a radius of 20 meters, to 12,9 mW if the event has a radius of 50 meters. With an expiration time of 120 seconds, the power

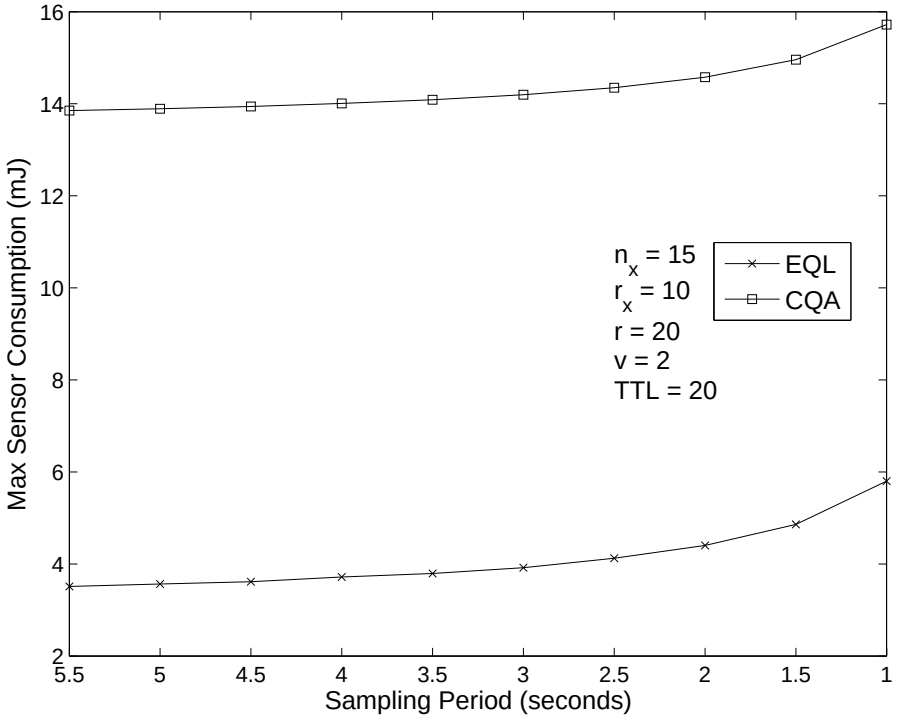


Figure 4.17: Maximum energy consumption of the sensors with increasing values of sampling rate of the sensors.

consumption of EQL is 5,3 mW, with an event of radius 20 meters, and it is 14,8 mW with an event of radius 50 meters. CQA, on the other hand, for every value of the expiration time, has a power consumption that ranges from 12,1 mW, if the event has radius 20 meters, to 68,3 mW, if the event has a radius of 50 meters.

Figure 4.15 shows that both systems are affected by the network density. However, this has a bigger impact on CQA than on EQL, since, also in this case, more sensors are involved in the event. In particular, with a number of neighbors per node of 13, the power consumption of EQL ranges from 2 mW, with a radius of the event of 12 meters, to 6 mW, with a radius of the event of 30 meters. The power consumption of CQA, with the same network density and the same event size, ranges from 5,4 mW to 25,5 mW. With a number of neighbors per node of 14, EQL has a power consumption of 2,2 mW, if the event has a radius 12 of meters, and of 6,4 mW, if the event has a radius of 30 meters. CQA has a power consumption that ranges from 5,7 mW to 27,3 mW. With a number of neighbors per node of 15, the power consumption of EQL is 2,4 mW, with an event of radius 12 meters, and it is 6,9 mW, with an event of radius 30 meters, while the power consumption of CQA, with the same network density and same event size, ranges from 6 mW to 29,1 mW.

We also analyze the energy consumption of the individual sensors and we report, in Figures 4.16, 4.17, 4.18, the total energy consumption of the sensor that consumes

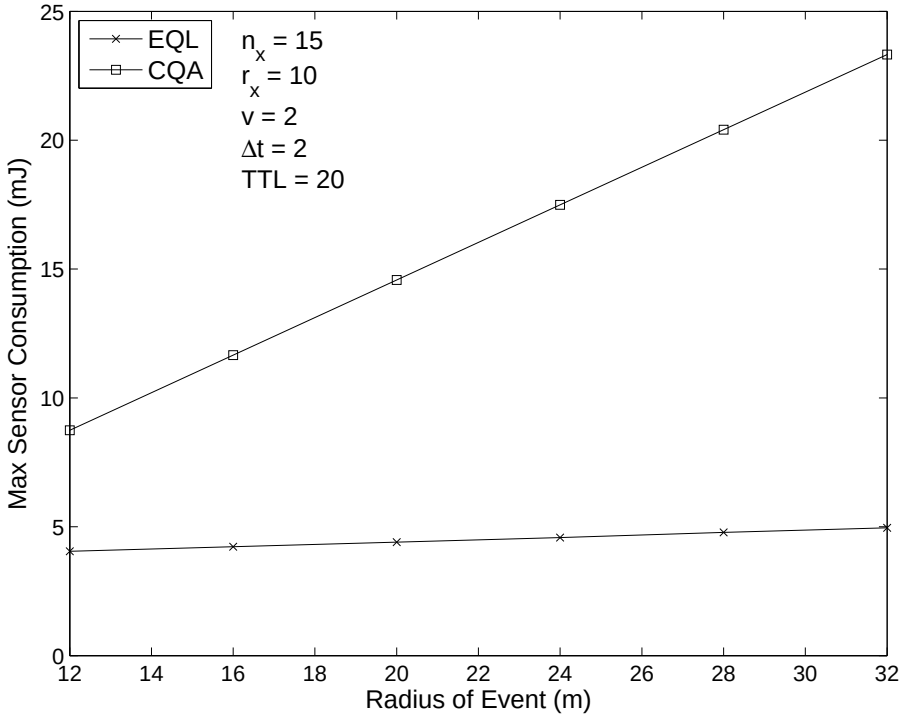


Figure 4.18: Maximum energy consumption of the sensors with increasing size of Event Area.

most energy in the scenarios where we increase the event speed, the sampling rate of the sensors and the size of the Event Area, respectively.

Figure 4.16 reports the maximum energy consumption of a sensor as the event speed increases. Despite what happens with the power consumption, that is independent from the time and thus increases with the event speed (see Figure 4.11), 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 figure shows also that a sensor in EQL has a 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 also notice 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, which corresponds to a decrease of 0.102 mJ/Km/h. Instead, the energy consumption in CQA, with the same values for the event speed, ranges from 14,6 mJ to 13.6 mJ, which corresponds to a decrease of 0.104 mJ/Km/h.

In Figure 4.17 we report the maximum energy consumption of a sensor with increasing value of the sampling rate of the sensors. Also in this case, EQL shows a smaller energy consumption, with respect to CQA, because of the smaller number of communications involved. However, in this case, the energy consumption of EQL grows faster than that of CQA. This happens because, in CQA, only the sensors in

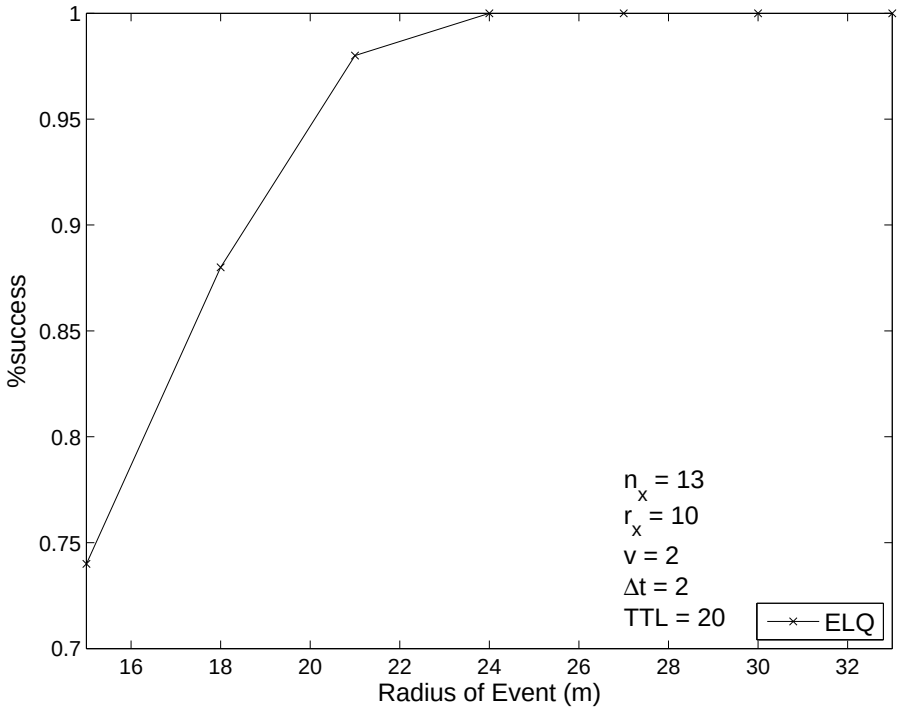


Figure 4.19: Percentage of successful tracking with increasing size of the Event Area.

the Event Area activate their transducers, since there is no sensor alert mechanism, 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 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, that is an increment of 0,508 mJ/s%.

Figure 4.18 reports the energy consumption of a sensor when the size of the event increases. This means that more sensors are involved in the event and, thus, each sensor is involved in more communications. CQA has a bigger energy consumption since, each time a new query is submitted, the sensors perform several transmissions to forward the query, while EQL executes only local communications in the border of the Event Area. This means that the energy consumption will grow more in CQA than in EQL as the size of the event increases. In fact, the energy consumption of EQL ranges from 4 mJ, with an event of radius 12 meters, to 5 mJ, with an event of radius 32 meters, that is an increment of 0,05 mJ/m. CQA, instead, has an energy consumption that ranges, for the same event size, from 8,7 mJ to 23,3 mJ, that is an increment of 0,73 mJ/m.

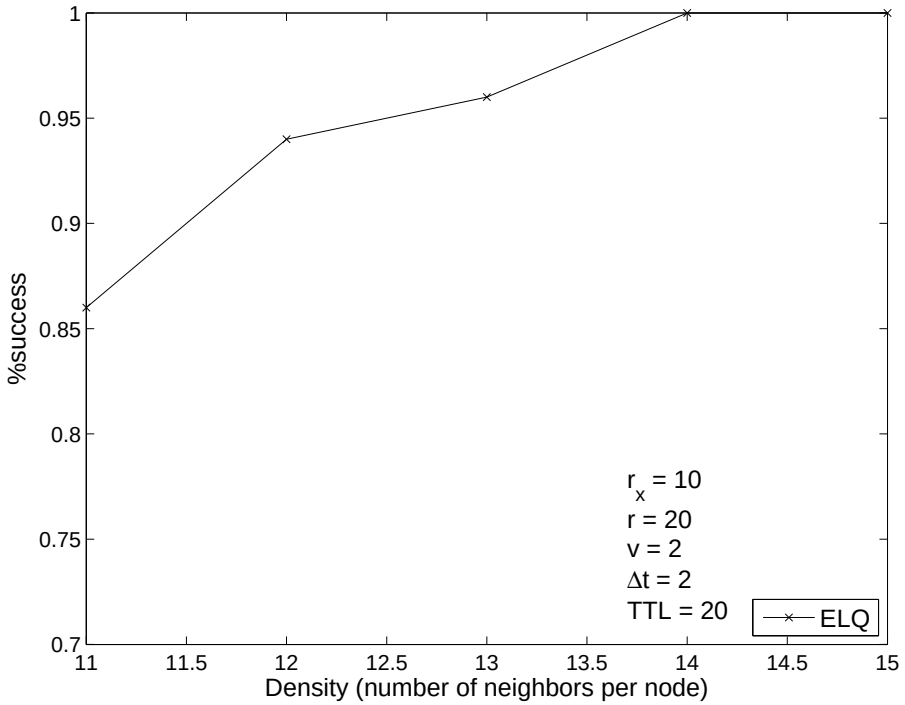


Figure 4.20: Percentage of successful tracking with increasing values of network density.

Success Rate

In this section we analyze the percentage of successfully tracked events (hereafter denoted with %success). The purpose of these experiments is to find a proper calibration of the parameters in order to ensure that the events are tracked successfully and with a low energy consumption.

The event can be lost because of the following reasons:

- 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;
- the event is too fast and the sampling rate too short, so the sensors do not detect it between two consecutive samplings;
- 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 different combinations of varying parameters.

Figures 4.19 and 4.20 show the %success with respect to the size of the Event Area and the network density, respectively. In these cases the event may be lost when traversing areas of the network in which there are no sensors (network holes). With a bigger Event Area, or a more dense network, the probability to lose the event

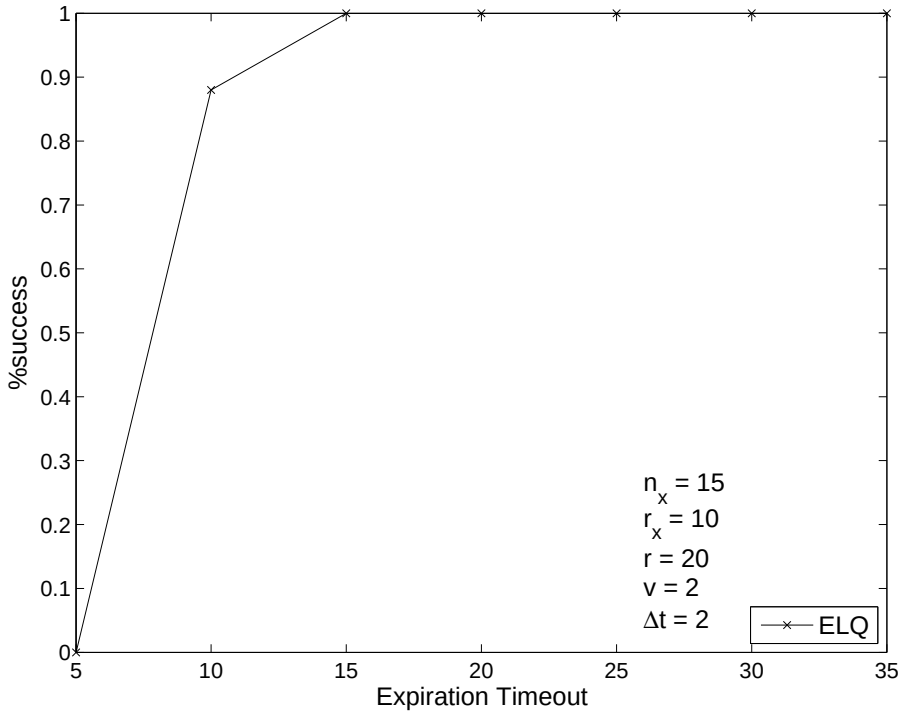


Figure 4.21: Percentage of successful tracking with increasing values of expiration time.

for this reason becomes lower and lower. In fact, we can see in Figure 4.19 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 its radius becomes bigger than 24 meters. Similarly, Figure 4.20 shows that the percentage of success with a number of neighbors per node equal to 11 is 86%, while it becomes 100% with a number of neighbors per node equal or greater than 14.

Figure 4.21 shows the %success for different expiration time values. 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; also we observe that the event is never lost with an expiration time equal or greater to 15 seconds. Obviously, 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 4.22 we analyze the %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 with a speed of at least 3 Km/h in order to be successfully tracked with a probability of more than 90%; with an expiration time of 15 seconds, the speed should grow to 6 Km/h in order to have a success rate

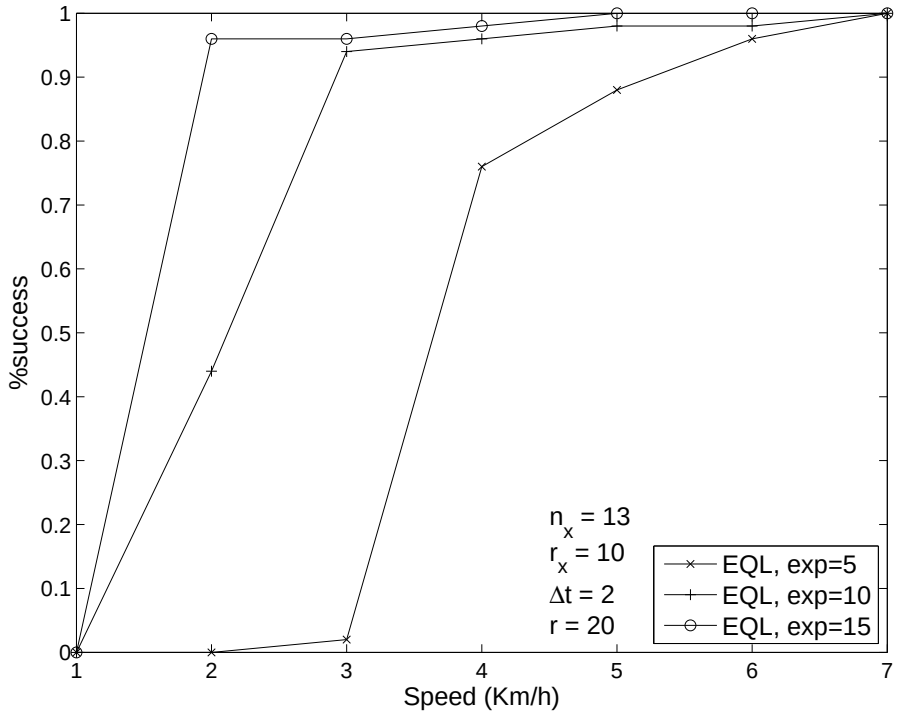


Figure 4.22: Percentage of successful tracking with increasing values of event speed and different values of expiration time.

greater than 90%. Therefore, the expiration time should be set according to the event speed, if this information is available.

The expiration time also affects the power consumption of EQL, since a high value for this parameter means that the alerted sensors, that are not involved in the event, remain monitoring for it for a longer time, thus wasting more energy. Figure 4.23 shows that a value of the expiration time greater than 700 leads to a power consumption higher than that of CQA (we recall that CQA is independent from the expiration time, because in that case the sensors are not alerted, meaning that the line related to CQA is flat). There is a trade-off between the percentage of successful tracking and the power consumption of EQL related to the expiration time. A larger value of this parameters ensures an higher success rate, but it also implies a greater power consumption of the sensors.

Figure 4.24(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 by. in this case the event is also 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

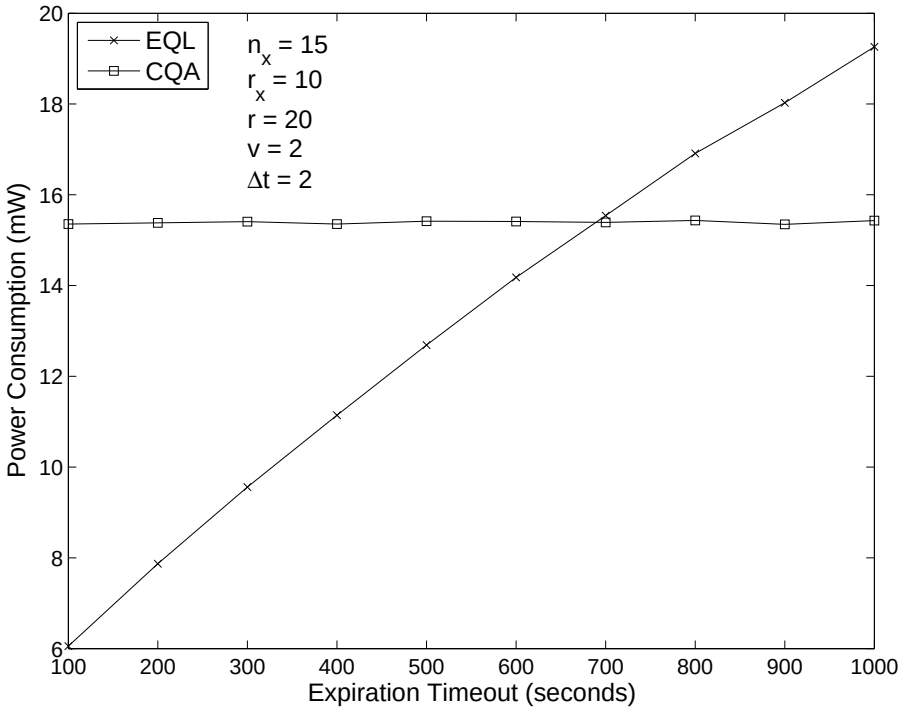
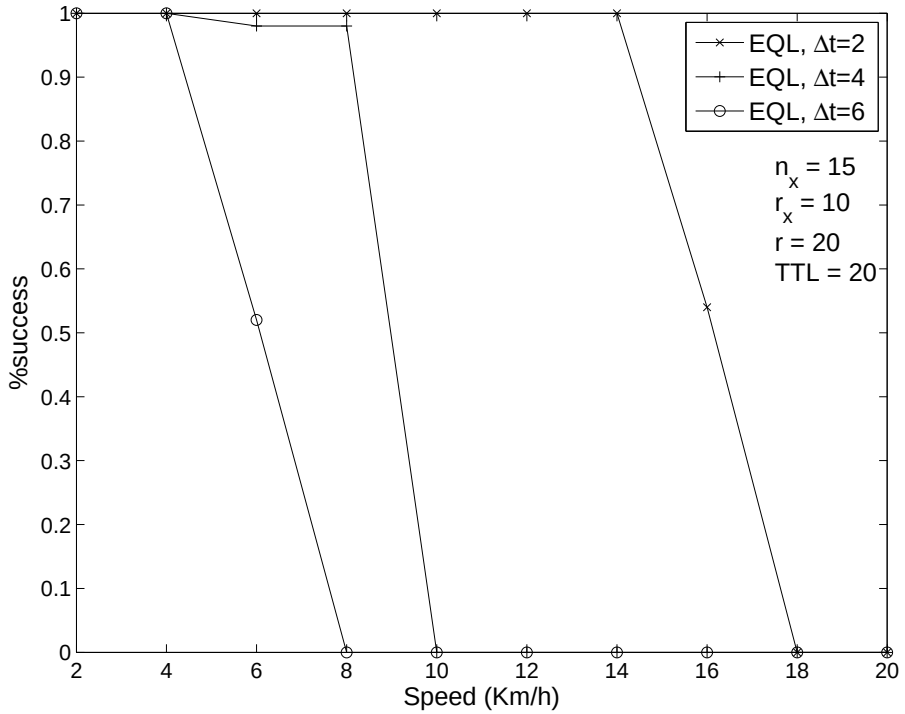


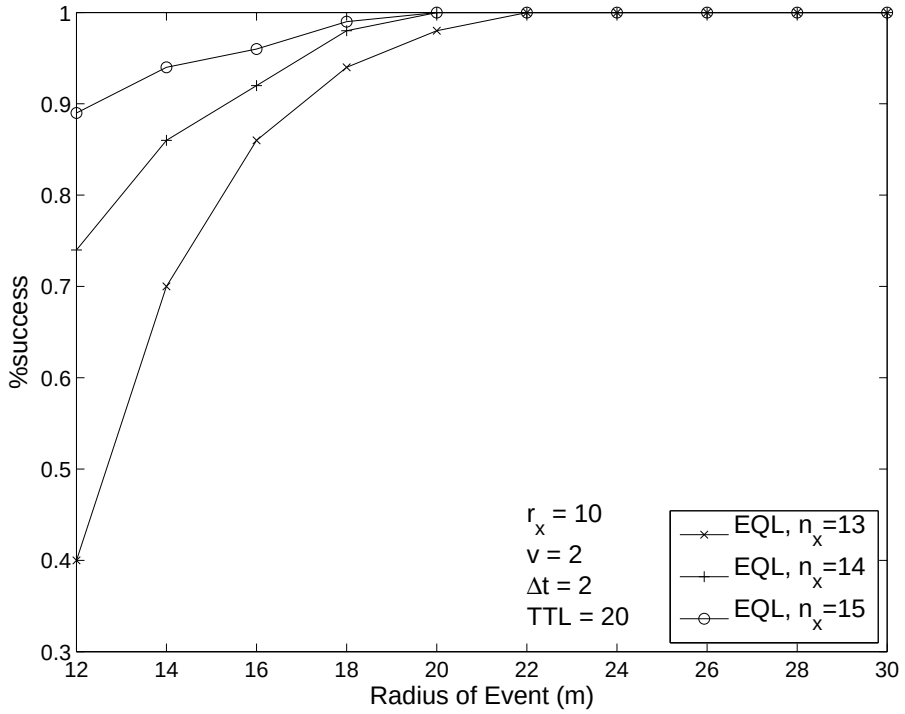
Figure 4.23: Power consumption with increasing value of the expiration time.

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. Similarly to the expiration time, the sampling rate should be also set according to the event speed, if available (and of course to the application requirements).

Finally, Figure 4.24(b) provides results for 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 a number of neighbors per node of 13, if the event has a radius of 12 meters it is successfully tracked only the 40% of times, while this percentage reaches 100% if the event has a radius of at least 22 meters. With a 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 a radius of 20 meters. The 100% of successful tracking is reached with an event of radius 20 meters also in case the number of neighbors per node is 15, but in this case the probability of not losing the event if it has a radius of 12 meters is almost 90%.



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



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

Figure 4.24: 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).

Chapter 5

Conclusions

WSN are an important enabling technology in several application fields, especially in tasks of detection of static events and tracking of mobile events. Although these tasks are common in WSN, their efficient implementation is still an issue, because it requires the execution of a number of low-level tasks distributed across a large number of low-power sensors. In this thesis we address these problems and we give two main contributions in terms of SQL-like languages and systems for the detection of static events, and for the detection and tracking of mobile events in WSN.

In particular, with respect to the detection of static events, we present MaD-WiSe, a system for data management in WSN that exploits a query language based on SQL to define a monitoring task that can be used to efficiently detect static events. MaD-WiSe was already implemented in a preliminary version at the beginning of this thesis, but it was significantly improved in the course of this thesis. Specifically, we have evaluated and implemented different strategies of query optimization, and we have introduced new cross-layer energy efficiency strategies. The introduction of these improvements have required an overall re-engineering of the MaD-WiSe architecture. The results of this work have been presented in [12], [13].

In the second part of the thesis we extended the approach proposed in MaD-WiSe to address the problem of tracking mobile events in WSN. Traditional query processing approaches are not efficient in tracking events, since movements or changes in the size or the shape of events require an update of the query, which in turn, implies an additional overhead to stop the outdated query and injection of the new one. Furthermore, traditional approaches do not consider the event as a data source, and, consequently, they are based on queries addressing individual transducers. On the other hand, by addressing the whole event as a source of data, it is possible to obtain higher-level information, such as speed and direction of the moving event. To this purpose we modeled the concept of composite event in WSN and we defined a new declarative language, EQL, that lets the user specify detection and tracking tasks of mobile, composite events. By means of an EQL query the sensors are instructed on how to cooperatively detect an event, and how to dynamically migrate the query in response to event movements. An EQL query can also specify the

high-level information related to the event (such as speed or direction) that need to be collected. The performance improvement in terms of overhead and scalability of EQL with respect to traditional approaches has been shown by simulations. The results of this activity have been presented in [62], [11].

The extension of EQL towards WSN where sensors may also be mobile is a short-term future direction of research. On the long term, future directions of research are the extension of this methodology for applications of ambient intelligence and web of things, thus also addressing different devices such as new generation smart-phones, intelligent appliances, domotic devices etc.

Acknowledgements

Questa tesi è il risultato di un lungo lavoro iniziato ben quattro anni fa! Ci sarebbero troppe persone da ringraziare, una alla volta, e ci vorrebbe un'altra tesi solo per quello.

Per cui mi limito a ringraziare la mia famiglia, che come al solito mi ha sempre sostenuto e mi è stata vicina in ogni momento; il Professor Stefano Chessa e il Dottor Giuseppe Amato, che mi hanno guidato e assistito pazientemente per tutta la durata del Dottorato; tutti i colleghi e amici di lavoro, gli amici d'infanzia, gli amici di vita, i compagni e amici delle varie squadre di calcetto che, anche senza volerlo o saperlo, mi hanno supportato durante questi anni e mi hanno aiutato ad arrivare in fondo a questa esperienza.

Bibliography

[1] http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf.

[2] TinyAODV implementation, TinyOS source code repository, http://cvs.sourcefourge.net/viewcvs.py/tinyos/tinyos-1.x/contrib/hsn/, 11/2009.

[3] MaD-WiSe: Management of Data in Wireless Sensor networks. http://mad-wise.isti.cnr.it.

[4] TinyOS. http://www.tinyos.net/.

[5] 2010. MEMSIC Powerful Sensing Solutions for a Better Life, http://www.memsic.com/company/about-memsic.html.

[6] T Abdelzaher, B Blum, Q Cao, Y Chen, D Evans, J George, S George, L Gu, T He, S Krishnamurthy, and et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *Proceedings of the International Conference on Distributed Computing Systems ICDCS*, pages 582–589, 2004.

[7] Michele Albano, Stefano Chessa, Francesco Nidito, and Susanna Pelagatti. Dealing with nonuniformity in data centric storage for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1398–1406, August 2011.

[8] Mohamed H. Ali. Phenomenon-aware sensor database systems. In *In Proc. of the EDBT Ph.D. Workshop*, pages 1–11, 2006.

[9] Giuseppe Amato, Paolo Baronti, and Stefano Chessa. Mad-wise: a distributed query processor for wireless sensor networks. In *Technical Report ISTI-2006-TR-39, Istituto di Scienza e Tecnologie dell'Informazione del CNR, Pisa, Italy*, page 39, 2006.

[10] Giuseppe Amato, Antonio Caruso, and Stefano Chessa. Application-driven, energy-efficient communication in wireless sensor networks. *Computer Communications*, 32(5):896–906, 2009.

[11] Giuseppe Amato, Stefano Chessa, Claudio Gennaro, and Claudio Vairo. Efficient detection of composite events in wireless sensor networks: Design and evaluation. In *IEEE Symposium on Computers and Communications (ISCC11).*, pages 821 – 823, Corfu, Greece, 2011.

[12] Giuseppe Amato, Stefano Chessa, and Claudio Vairo. Optimizing network-side queries with timestamp-join in wireless sensor networks. In *35th Annual Conference of IEEE Industrial Electronics (IECON09).*, pages 2653 – 2658, Porto, Portugal, 2009.

[13] Giuseppe Amato, Stefano Chessa, and Claudio Vairo. MaD-WiSe: A distributed stream management system for wireless sensor networks. *Software Practice & Experience*, 40(5):431–451, 2010.

[14] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD Conference, Baltimore Maryland*, pages 13–24, 2005.

[15] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. Fad, a powerful and simple database language. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *13th International Conference on Very Large Data Bases (VLDB), Brighton, England*, pages 97–105. Morgan Kaufmann, 1987.

[16] Paolo Baronti, Prashant Pillai, Vince Chook, Stefano Chessa, Alberto Gotta, and Y. Fun Hu. Wireless Sensor Networks: a Survey on the State of the Art and the 802.15.4 and ZigBee Standards. *Computer Communications*, 30:1655–1695, 2007.

[17] Eduardo Cañete, Manuel Díaz, and Bartolomé Rubio. A wireless sensor network framework based on light databases. *Software Practice & Experience*, 42(7), 2012.

[18] Carlos T. Calafate, Carlos Lino, Juan-Carlos Cano, and Pietro Manzoni. Modeling emergency events to evaluate the performance of time-critical WSNs. In *Proceedings of the The IEEE symposium on Computers and Communications*, ISCC '10, pages 222–228, Riccione, Italy, 2010. IEEE Computer Society.

[19] Antonio Caruso, Stefano Chessa, Swades De, and Alessandro Urpi. Gps free coordinate assignment and routing in wireless sensor networks. In *Proceedings of IEEE INFOCOM, Miami FL*, pages 150–160, 2005.

[20] Wang-Rong Chang, Hui-Tang Lin, and Zong-Zhi Cheng. Coda: A continuous object detection and tracking algorithm for wireless ad hoc sensor networks. In *5th IEEE Consumer Communications and Networking Conference CCNC*, pages 168–174, Las Vegas, NV, 2008.

[21] Chee-Yee Chong, Feng Zhao, S. Mori, and S. Kumar. Distributed tracking in wireless ad hoc sensor networks. In *Proceedings of the Sixth International Conference of Information Fusion*, pages 431–438, Cairns, Queensland, Australia, 2003.

[22] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[23] Soledad Escolar, Stefano Chessa, and Jesús Carretero. Cross-layer optimization of low power listening mac protocols for wireless sensor networks. *IEEE Symposium on Computers and Communications*, 0:684–691, 2011.

[24] C R Farrar, S W Doebling, and D A Nix. Vibration-based structural damage identification. *Philosophical Transactions of the Royal Society A Mathematical Physical and Engineering Sciences*, 359(1778):131–149, 2001.

[25] Paolino Di Felice, Massimo Ianni, and Luigi Pomante. A spatial extension of TinyDB for wireless sensor networks. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 1076–1082, Marrakech 2008.

[26] David Gay, Philip Levis, David Culler, and Eric Brewer. *NesC 1.1 Language Reference Manual*, May 2003. http://nescc.sourceforge.net.

[27] C. Gomez, P. Salvatella, O. Alonso, and J. Paradells. Adapting aodv for ieee 802.15.4 mesh sensor networks: Theoretical discussion and performance evaluation in a real environment. In *International Symposium on on World of Wireless, Mobile and Multimedia Networks (WoWMoM, Buffalo-Niagara Falls NY)*, pages 159–170, 2006.

[28] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah 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*, pages 56–67, 2000.

[29] ISTI-CNR, Via G. Moruzzi, 1, 56124, Pisa, IT. *SensorViz/MaD-WiSe*, version 1.3 edition, July 2006. http://www.di.unipi.it/∼ste/MaD-WiSe/manual_13.pdf.

[30] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGARCH Comput. Archit. News*, 30(5):96–107, 2002.

[31] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4), September 2007.

[32] Krasimira Kapitanova, Sang H. Son, and Kyoung-Don Kang. Using fuzzy logic for robust event detection in wireless sensor networks. *Ad Hoc Networks*, 10(4):709–722, June 2012.

[33] Thomas Kleinberger, Martin Becker, Eric Ras, Andreas Holzinger, and Paul Müller. Ambient intelligence in assisted living: enable elderly people to handle future interfaces. In *Proceedings of the 4th international conference on Universal access in human-computer interaction: ambient interaction*, UAHCI'07, pages 103–112, Berlin, Heidelberg, 2007. Springer-Verlag.

[34] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[35] Sunil Kumar, Kashyap K. R. Kambhatla, Bin Zan, Fei Hu, and Yang Xiao. An energy-aware and intelligent cluster-based event detection scheme in wireless sensor networks. *International Journal of Sensor Networks*, 3(2):123–133, February 2008.

[36] Chih-Yu Lin, Wen-Chih Peng, and Yu-Chee Tseng. Efficient in-network moving object tracking in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 5(8):1044–1056, 2006.

[37] Carlos Lino, Carlos T. Calafate, Arnoldo Diaz-Ramirez, Pietro Manzoni, and Juan-Carlos Cano. Studying the feasibility of IEEE 802.15.4-based WSNs for gas and fire tracking applications through simulation. In *11th IEEE International Workshop on Wireless Local Networks (LCN)*, pages 875–881, Bonn, Germany, 2011.

[38] Kebin Liu, Lei Chen, Yunhao Liu, and Minglu Li. Robust and Efficient Aggregate query processing in wireless sensor networks. *Mobile Networks and Applications*, 13(1-2):212–227, 2008.

[39] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA*, 2002.

[40] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

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

[42] Navneet Malpani, Jennifer L. Welch, and Nitin Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, DIALM '00, pages 96–103, New York, NY, USA, 2000. ACM.

[43] Dónall McCann and Mark Roantree. A query service for raw sensor data. In *EuroSSC*, volume 5741 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2009.

[44] Sun Microsystem. Jdbc: Java database connectivity. http://java.sun.com/jdbc.

[45] M.F. O'Connor, K. Conroy, M. Roantree, A.F. Smeaton, and N.M. Moyna. Querying xml data streams from wireless sensor networks: an evaluation of query engines. In *Third International Conference on Research Challenges in Information Science (RCIS), Fes, Morocco*, pages 22–30. IEEE, 2009.

[46] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD Conference, San Diego California*, pages 563–574, 2003.

[47] Elizabeth Olule, Guojun Wang, Minyi Guo, and Mianxiong Dong. Rare: An energy-efficient target tracking protocol for wireless sensor networks. In *International Conference on Parallel Processing Workshops ICPPW*, pages 76–81, Xian, China, 2007. IEEE Computer Society.

[48] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, January 2005.

[49] C. Perkins and E. Belding-Royer. Ad hoc on demand distance vector routing. In *Mobile Computing Systems and Applications (WMCSA), New Orleans LA*, pages 90–100, 1999.

[50] A V U Phani Kumar, Adi Mallikarjuna Reddy V, and D. Janakiram. Distributed collaboration for event detection in wireless sensor networks. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, MPAC '05, pages 1–8, New York, NY, USA, 2005. ACM.

[51] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 95–107, New York, NY, USA, 2004. ACM.

[52] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. Design considerations for solar energy harvesting wireless embedded systems. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[53] V. Solai Raja and S. S. Sreeja Mole. A predictive energy-efficient mechanism to support object-tracking sensor networks. *IJCA Proceedings on International Conference in Recent trends in Computational Methods, Communication and Controls (ICON3C 2012)*, ICON3C(8):13–17, April 2012. Published by Foundation of Computer Science, New York, USA.

[54] Ross Rosemark, Wang-Chien Lee, and Bhuvan Urgaonkar. Optimizing energy-efficient query processing in wireless sensor networks. In *8th International Conference on Mobile Data Management (MDM), Mannheim, Germany*, pages 24–29, 2007.

[55] Shad Roundy, Dan Steingart, Luc Frechette, Paul K. Wright, and Jan M. Rabaey. Power sources for wireless sensor networks. In *European Conference on Wireless Sensor Networks (EWSN)*, pages 1–17, 2004.

[56] Ghalib Shah, Muslim Bozyigit, and Demet Aksoy. Adaptive pull-push based event tracking in wireless sensor actor networks. *International Journal of Wireless Information Networks*, 18:24–38, 2011. 10.1007/s10776-010-0126-9.

[57] Mark Stemm and Randy H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices, 1997.

[58] Maneesha Sudheer. *Wireless Sensor Network for Disaster Monitoring.* Yen Kheng Tan (Ed.), 2010.

[59] Egemen Tanin, Songting Chen, Junichi Tatemura, and Wang-Pin Hsiung. Monitoring moving objects using low frequency snapshots in sensor networks. In *MDM '08: Proceedings of the The Ninth International Conference on Mobile Data Management*, pages 25–32, Washington, DC, USA, 2008. IEEE Computer Society.

[60] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, pages 1–8, New York, NY, USA, 2003. ACM.

[61] Niki Trigoni, Yong Yao, Alan Demers, Johannes Gehrke, and Rajmohan Rajaraman. Multi-query optimization for sensor networks. In *International Conference on Distributed Computing in Sensor Systems (DCOSS), Marina del Rey CA*, pages 307–321, 2005.

[62] Claudio Vairo, Giuseppe Amato, Stefano Chessa, and Paolo Valleri. Modeling detection and tracking of complex events in wireless sensor networks. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 235–242, Istanbul, Turkey, 2010.

[63] Sudarshan Vasudevan, Jim Kurose, and Don Towsley. Design and analysis of a leader election algorithm for mobile ad hoc networks. In *Proceedings of the 12th IEEE International Conference on Network Protocols*, ICNP '04, pages 350–360, Washington, DC, USA, 2004. IEEE Computer Society.

[64] Chinh T. Vu, Raheem A. Beyah, and Yingshu Li. Composite event detection in wireless sensor networks. *21st IEEE International Performance, Computing, and Communications Conference.*, 0:264–271, 2007.

[65] Markus Wälchli, Samuel Bissig, Michael Meer, and Torsten Braun. Distributed event tracking and classification in wireless sensor networks. *Journal of Internet Engineering*, 2(1):117–126, 2008.

[66] Markus Wälchli, Piotr Skoczylas, Michael Meer, and Torsten Braun. Distributed event localization and tracking with wireless sensors. In *WWIC '07: Proceedings of the 5th international conference on Wired/Wireless Internet Communications*, pages 247–258, Berlin, Heidelberg, 2007. Springer-Verlag.

[67] H. Yang and B. Sikdar. A protocol for tracking mobile targets using sensor networks. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, pages 71–81, Anchogare, AK, 2003.

[68] Yinying Yang, Arny Ambrose, and Mihaela Cardei. Coverage for composite event detection in wireless sensor networks. *Wireless Communications and Mobile Computing*, 11(8):1168–1181, August 2011.

[69] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[70] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *CIDR, Asilomar CA*, 2003.

[71] Wei Ye, John Heidemann, and Deborah Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*, 12(3):493–506, June 2004.

[72] Xingbo Yu, Koushik Niyogi, Sharad Mehrotra, and Nalini Venkatasubramanian. Adaptive middleware for distributed sensor environments. *IEEE Distributed Systems Online*, 4(5):–, May 2003.

[73] Shu Zhou, Min-You Wu, and Wei Shu. Improving mobile target detection on randomly deployed sensor networks. *International Journal of Sensor Networks*, 6(2):115–128, October 2009.

[74] Zheng Zhou and Gang Qu. An energy efficient adaptive event detection scheme for wireless sensor network. *Application-Specific Systems, Architectures and Processors, IEEE International Conference on*, 0:235–238, 2011.

[75] Xianjin Zhu Xianjin Zhu, H Gupta, and Bin Tang Bin Tang. Join of multiple data streams in sensor networks, 2009.