State of the Art and Future Directions in Wireless Sensor Network's Data Management

Project:
Management of Data in Wireless Sensor Networks (MaD-WiSe)

Deliverable 1

G. Amato, A. Caruso, S. Chessa, V. Masi and A. Urpi

Abstract

This report presents the results of the first three months of activity of the project Management of Data in Wireless Sensor Networks (MaD-WiSe). More specifically, it presents a survey on the state of the art on Wireless Sensor Networks, focusing on the issues related to communication, data centric paradigms and stream processing applied to sensor networks.

Categories and subject descriptors: C.2.1 Wireless communication - C.2.1 Network Architecture and Design - C.2.2 Network Protocols - I.4.8 Sensor fusion - I.2.9 Sensors - C.2.4: Distributed databases - H.3.3 Information Search and Retrieval.

1 Introduction

A Sensor Network [2] is a specialized ad hoc network composed of a large number of sensors, which are low power, low cost nodes. A sensor comprises one or more sensing units, a processor and a radio transceiver, and it is powered by an embedded battery. Sensors collect information about the surrounding environment (sensor field) and they are interfaced with external sink nodes that issue queries about sensed data to the network as a whole. Typical applications of sensor networks are environment sampling, disaster areas monitoring, health monitoring, surveillance, security, inventory management, and they have also been envisioned as an architectural support for applications of ubiquitous/pervasive computing [8].
In the effort of improving the management of data produced by sensor networks, it has recently been proposed to integrate database and sensor network technologies [41, 40]. This approach would provide a more effective analysis and exploitation of sensed data, for better detection of situations leading to risk/security issues, thus providing a realistic real-time emergency management.
The integration of these two technologies, however, requires a new vision of sensor network and database approaches. Sensors produce continuous, possibly infinite, streams of data. Simple (one column) streams of data can be combined into streams of tuples. However, data streams produced by the sensors hardly adapt to data model of the traditional database technologies.
One important difference is that traditional database technology deals with almost static data sets as compared to data managed in sensor networks. Even if traditional database technology can deal with evolving data sets, the evolution rate of data produced in a sensor network is much higher than that typically considered in traditional databases systems. Besides, in traditional database technology data are stored in a persistent repository and that can be accessed at any time. Query languages, query optimization strategies, and query processing strategies are based on this assumption. Furthermore, data processed by a sensor network should be immediately consumed, and query processing should react in real time to deal with sensed data. Data streams produced by the sensors might be processed by (evolutes, smart) sensors themselves,

but, in this respect, strategies of query optimization and query processing should be redefined. Finally query languages should reflect this new way of handling data.
This integration poses new challenges also to the sensor network architecture. In fact, the state of the art architectures of sensor networks are quite unfeasible to support even simple database operations. For this reason new proposals of enhanced sensor networks appeared in the recent literature [9, 41], which, in the essence, still consider the network as an appliance permanently connected to the sink node. This assumption does not fit with our vision, in which the sensor network is able to autonomously process complex queries originated by the sink node. When a connection with the network is available, the sink receives responses related to previous queries, and originates new queries which can be related to the past as well as to the future sensed data. To support this vision, the sensor network should be able to store and process the sensed data autonomously of the sink node.
This deliverable presents a survey of the state of the art on sensor networks and database technology applied to sensor network, and discuss the directions we intend to investigate in order to integrate database and sensor network technologies.
The rest of the document is organized as follows. Section 2 analyzes the sensors architecture, describing the hardware used in off the shelf sensors (2.2), a simple but powerful operating system (2.3), a routing algorithm which is imposing itself as a standard (2.4) and a technique that can be used in an eventual middleware layer (2.5). Section 3 analyzes the problem of query management and optimization, with respect to classical databases (3.1) and streams (3.2), describing the state of the art in database management over sensor networks (3.3). Conclusions and future works are left to Section 4.

2 Architectural issues in sensor networks

Current advances in CMOS technology enable the development of sensors embedding some computational and communication commodities. While dramatically constrained with respect to other wireless mobile systems like ad hoc networks, it is possible to organize many sensors in a network. In this way, huge sets of measurements may be easily available for analysis, monitoring and research.
The most precious resource in such a system is the energy: for this reason is necessary to design energy efficient communication and computation strategies (not necessarily disjoint). This requirement pushed researchers in the direction of *data centric* strategies: communications are in general not relative to nodes, but to data. In a few words, the network is designed in order to store an retrieve data (or to publish data streams and subscribe to them), regardless of which nodes need or provide them.
Another key concept in the design of sensor networks is robust fault tolerance. If a node fails (e.g. for hardware crashes, energy exhaustion or just accidental movement), all the operations in the networks must continue normally, because it is unlikely that the faulty node will be repaired.
This document organization reflects the layer architecture of sensors. In Section 2.1 is given a brief description of a base model of sensor networks. In Section 2.2 is analyzed the hardware composing the MOTES, which are a real world implementation of sensors. The concepts behind their operative system (called TinyOS) are described in Section 2.3.
A geographic routing, the de facto standard (at least in many simulations) used by many data localization techniques, is reviewed in section 2.4.
Data centric diffusion and storage mechanisms for sensor networks are analyzed in Section 2.5, leaving to Appendix 4 a description of analogous approaches adopted in the peer to peer community.

2.1 The model

We describe here a basic model of sensor networks with a few variants, useful for the purposes of this document. A broader survey is presented in [1].
A sensor network is composed by a large amount of sensors (also called nodes) randomly spread over a geographical area (sensing field). It is typically assumed the sensors are uniformly distributed with a fixed density in the field, but they can be manually disposed to form fixed topologies. Each sensor is composed at least by the following components:

- one or more sensing devices, used to measure environmental data,

- a processor and a small amount of memory,
- a low power broadcast[1] transmitter, like a radio transceiver equipped with an omnidirectional antenna.

As in the ad hoc networking paradigm, nodes have to cooperate in order to enable multiple hops communications.

In general sensors do not move, or have a very low mobility, and they are very prone to failures, due to hardware crashes, energy exhaustion or environmental interferences. If the sensor network applications require data to be associated to the position they were measured in, nodes should be aware of their position in the network (by means of an absolute or relative coordinate system)

Sensors are in general heterogeneous regarding their functionality (e.g. temperature sensors, visual sensors, etc) and capacity (computation/memory) In the latter case it is possible to design protocols that focus on an efficient usage of low capacity nodes, leaving all the heavy duties to high capacity ones. In the former case, every single operation and communication must be optimized.

Sensors may be instructed to periodically take measures and perform data preprocessing. Typical tasks may be:

- send the data, or a combination of data (average, sum, etc), to some interested node,
- store the data (which may be acquired later in time),
- send a message as a reaction to some observed event.

In the classical model there is at least one special node, called the sink, which has more capacity than sensors. It may be thought as not energy constrained, and it may/may not communicate with a single broadcast with all the nodes composing the network. Operation requests originate in large part at the sink, and data flows in its direction.
Current approaches, however, aim at designing autonomous sensor networks, i.e. networks in which one or more sinks periodically connect, eventually distributing tasks and collecting data. It is then vital to develop efficient memorization mechanisms, which allows nodes to efficiently distribute data for storage and the sink to collect the data without querying all the nodes.
It is clear that, given the stringent constraints of sensor networks, the application (and the networking layers) should be tuned to the specific case the network is needed for. For example, routing and/or topology control could be designed taking into account the application needs, in order to save energy.
It is generally accepted that a data centric approach can lead to better results in terms of application complexity and network performances. Following this approach, nodes do not have an identifier known to all the network, but may be seen as labelled by the data they process. Thus, mechanisms to reach the data (and not the nodes) must be offered, detaching sensors from operations they perform, and introducing a better degree of fault tolerance at lower levels in the software layers. Data diffusion ([5]) and data centric storage ([22, 19]) will be analyzed in Section 2.5.

2.2 Hardware

We analyze the Motes architecture, presented in [6]. Since the Motes are sensors effectively developed and used, they represent in some way the state of the art in the field, as well as a practical test bed for protocols and sensor oriented applications.
A mote is composed by a MICA board (Figure 1(a)), including CPU and radio, and a sensor board (like the MTS310CA shown in Figure 1(b)) offering several sensing facilities (the MTS310CA can measure light, sound, temperature, 2 dimensional acceleration and 2 dimensional magnetic fields).

The MICA is equipped with an 4/8Mhz RISC processor ATMEL ATmega 103L. This processor has 32 general registers, 128 Kb (512 pages of 256 bytes each) of flash used as program memory and 4Kb of SRAM used as data memory (expandable with 64K of external SRAM). It also has a 4 Mbit Flash used as external memory with the help of a coprocessor.

[1]The broadcast nature of the transmitter is to ensure that communications will be possible without infrastructures and in absence of a topological ordering of the nodes.

(a) The MICA board (b) The MTS310CA board

Figure 1: Motes components.

The instruction set is composed by 121 instructions (the majority executed in a single clock cycle, with pipelined fetch/exec), but no floating point support. Concerning energy management, the processor has the following states:

- active,
- idle, i.e. with the CPU stopped but the SRAM, timer, interrupt logic working,
- power down, i.e. shut down with only the interrupt logic on,
- power save (same as power down, but with an asynchronous timer running).

The MICA board is equipped with a RFM TR1000 916 Mhz radio transceiver (other frequencies are available, given interferences with GSM). It is the lowest power radio in commerce at the moment, and it operates in OOK (10 kb/s) or ASK (115 kb/s) mode, with a transmission range influenced by environmental conditions (typical measured value is of the order of 10 meters). The radio is likely to be changed in future Motes generations, because of its poor performances (moreover, as reported in [40], it uses about 4 μJ per bit transmitted, enabling the transmission of only about 14 MB with the current state of the art micro batteries.

Finally, it has three LEDS used as output device and a serial port for asynchronous data transfer (at the byte level).

2.3 OS

On the top of the MOTES hardware (Section 2.2), a barebone operative system called tinyOS has been developed. The system was kept to the minimum because of the tight space constraints of the architecture and because many aspects like networking and sensors management are heavily application dependant.
The system is composed by a processor initialization routine (172 bytes), a small C runtime (82 bytes) and an event scheduler (178 bytes) which is the core of the tinyOS. Several higher level modules (e.g. MAC layer implementations, routing, etc) are freely available.
A complete application is a graph of components: each component represent a software module (which can be used by or use other modules), and is composed by:

- a set of command handlers,
- a set of event handlers,
- an encapsulated fixed size memory frame,
- a set of tasks.

Basically, for a given component, command handlers represent the exported interface, while event handlers are the interface to get the result of computation carried on in other components. The execution of commands (i.e. operations carried on by underlying components) is asynchronous, and raises events when completed or aborted.
Normally, commands are called in tasks, which may be seen as the application threads. In order to keep

complexity at the minimum, tasks and handlers run until their completion, and may be preempted just by hardware interrupts, returning to execution after the interrupt service. This also enables the static allocation of memory at compile time.
The language used to write applications is a C dialect, called nesC ([14]), and a powerful Mote emulator with networking simulator capabilities ([13]) is also provided.

2.4 Routing

In this section we analyze the evolution of routing algorithms used in sensor networks.
Routing algorithms used in wired networks (e.g. Distance Vector, Link State and Path Vector) do not perform well in wireless and mobile networks, given the transitory nature of links and the high number of failures. Moreover, they require big routing tables, wasting a precious resource in sensors, and accurate information about the state of other routers, which is very expensive to obtain when nodes move.
Algorithms introduced for ad hoc networks (e.g. AODV [15] and DSR [7]) trade the optimality of routing with the opportunity of establishing routes only when needed. With cashing of routes, a good compromise is reached, avoiding the generation of new messages every time the same route is used.
In the case of sensor networks, both routes maintenance and routes caching should be kept as low as possible. The former process, in general, causes an energy waste due to frequent messages routers (i.e. nodes) have to exchange, while the latter requires storage capabilities that in a sensor are limited and precious to keep recorded data.
We start by describing the data diffusion proposal, that merges routing (usually from or towards sink) and topology maintenance, reserving more attention to the GPSR routing algorithm, which is increasingly used in sensor netowrk given its generality and scalability.

Data Diffusion

Data diffusion ([5]) is one of the first proposals in which the focus was moved from communications between a source and a destination to communications relative to some data.

The scenario fitting the data diffusion model is the following: nodes produce data, and they must be delivered to all the interested nodes. Classic techniques to solve this problem are flooding (i.e. the data is sent to all the neighbors, which in turn do the same) or gossiping (a subset of neighbors is randomly chosen to forward). The former technique always reach all the interested nodes, but it is extremely expensive, while the latter does not give any guarantees about successful delivery.

Alternatively, any node interested in some data should to discover the source of them (or the location where they are taken). Once the data source is discovered, a communication channel is established and sensed data are routed though this channel with a given rate. Multiple channels must be established through the same nodes, thus wasting resources. If more nodes are responsible to take the same measures, their data may overlap, causing again an energy (and a bandwidth) waste.

For this reason, in [5] it is proposed to uniquely label all the sensed data with some meta-data, with a succinct representation in terms of space. For example, if all nodes have a unique id, the meta-data $<$ id $>$ may be used to describe all the data collected by sensor id. A meta data (x, y) may represent all the data collected at a given location, or a tuple $(x, y, type)$ is data of a given type (with data types opportunely coded) taken in a precise position. However, authors do not introduce a standard way to label data: every application may choose the schema that best fits to the considered case. Intuitively, a meta-data is an efficient (i.e. compact) way to advertise or to query some specific data.

The data diffusion approach uses a publish/subscribe approach: data producer publish their activity sending ADV messages to all their neighbors. ADV messages contain the description (i.e. the meta-data) of the produced data. Then, neighbors interested (or which received a request for that type of data) reply with a REQ message, and the data is effectively sent. Figure 2 (from [5]) shows a simple example of data diffusion.

This technique may be used, in case of large amounts of data (like periodically taken measures) in order to build a spanning tree that covers all the interested nodes. When a component of the tree disappear, a new ADV/REQ round may be run for the creation of a new tree. When a node not in the tree needs to receive the data, it sends a REQ that reaches the nearest node already in the tree, and the new path is added.
Authors present also an energy aware version of the protocol: if a node energy drops under a threshold, the

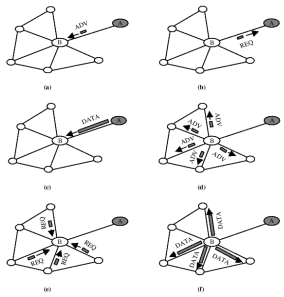


Figure 2: Data diffusion.

node reduces its participation to the protocol (i.e. it does not forward).
Simulations show that the solution is a good compromise between the speed of data dissemination (higher with the flooding technique) and the energy consumption.

Position awareness

As pointed out in Section 2.1, sensors are often aware of their geographical position, and this feature may be exploited to reason in terms of final destination position instead of routes.
Normally it is assumed that the information about location is obtained by means of a GPS system. GPS, however, may be too expensive in case of very large networks, or difficult to adopt in case of indoor systems. Other solutions are analyzed in [4].
In [20] is presented a location diffusion method that overcomes to these problems. Authors assume a very large number of nodes uniformly distributed on a rectangular area. They are interested in building a virtual coordinate system that can be used by a greedy geographic routing algorithm. It is not a concern of the paper to build a correspondence between virtual and real space.
In a complexity climax, the following scenarios are considered:

- nodes situated on the network border know their location (i.e. geographic coordinates) while others do not,
- nodes situated on the border just know they are on the border, while others do not know anything,
- no information about location is present in any node.

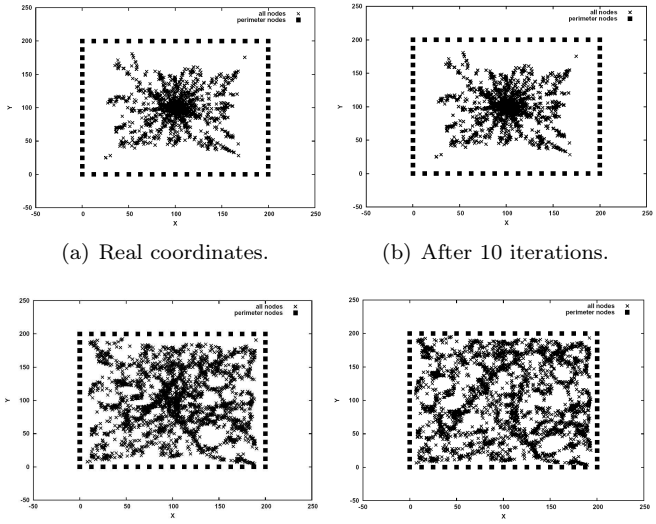
In the first case, all the node that are not located on the network border start by assuming they are on the center of the network, and iteratively compute their bidimensional coordinates in the following way:

$$x_i = \frac{\sum_{j \in N_i} x_j}{\#N_i}$$

$$y_i = \frac{\sum_{j \in N_i} y_j}{\#N_i}$$

In the previous formulas, N_i is the set of neighbors of node i, while $\#N_i$ is the cardinality of the set.
Simulations shown that after 1000 iterations, a routing success rate higher than 0.99 is obtained (better than in the case of true coordinates), with an average path length just slightly worse than in the optimal case.
The high number of iterations necessary to achieve high success rate depends on the choice of the central point as initial coordinate for all the network nodes. Figures 3(a)-3(d) (borrowed from [20]) shows a real scenario, and how virtual coordinates approximate reality after 10, 100 and 1000 iterations.

Somehow paradoxically, the number of iterations needed to achieve a good success rate can be lowered in the second presented scenario. If nodes sitting on the borders of the network know they are there, without knowing their position, authors propose a protocol to distributely build virtual coordinates based on relative hop distances between them. Border nodes sequentially flood the entire network with a beacon. In this way, all the nodes in the network learn the relative distances between border nodes (when the protocol ends). It



(a) Real coordinates. (b) After 10 iterations.

(c) After 100 iterations. (d) After 1000 iterations.

Figure 3: Iterative process.

is possible to compute the coordinates of border nodes by choosing x_is and y_is that minimize the following formula:

$$\sum_{i,j \in \text{perimeter}_\text{s}\text{et}} (\text{hopDist}(i,j) - \text{dist}(i,j))^2$$

where hopDist(i,j) is the measured hop count between node i and node j, while dist(i,j) is the Euclidean distance between the virtual coordinates of i and j.
After all nodes have computed virtual coordinates for the ones standing on the perimeter, the same protocol described in previous scenario is started. Now nodes know their distance from perimeter nodes, and so may choose better initial coordinates than the central point. In fact, the same success rate (i.e. over 99%) may be reached with just 10 iterations (after the initial set up).
In order to avoid inconsistent computations caused by packet loss, a bootstrapping technique is used: two special nodes start flooding the network with HELLO messages before the first phase of the protocol. All the following steps will include also the bootstrap nodes coordinates, which can be thought as defining the x and y axis.
Finally, if no information at all about positions is available, authors propose an extension to the protocol in which nodes assume to be on the border if none of their neighbors has a higher hop distance from bootstrap nodes[2]. After this step, the same protocol as explained may start.
Authors provide a large number of simulated scenarios, showing that the approach performs well in presence of obstacles (i.e. holes in the topology) and even with low mobility.

An overview of GPSR routing

In [10], authors present a simple algorithm that allow nodes to route packets to a node positioned in a given location, that requires, for every node, only the knowledge of theirs neighbors positions. The amount of stored data is then proportional to nodes density and not to nodes number, and it needs to be updated with a frequency that increases with nodes mobility (in general very low in sensor networks).
Nodes using GPSR then periodically broadcast their position (represented by two 64 bits IEEE floating point numbers) to neighbors, and keep a table of neighbors positions. It is possible to reduce the number of beacons with a piggybacking technique.
When a packet must be routed towards a given point, a node chooses the neighbor that have minimum distance from the destination. Figure 4 (from [10]) shows a simple case in which node x, having to route a

[2]Authors use second step neighbors, in order to have a better solution.

packet to D, chooses y as next hop. There may be cases in which no neighbors are closer to the destination

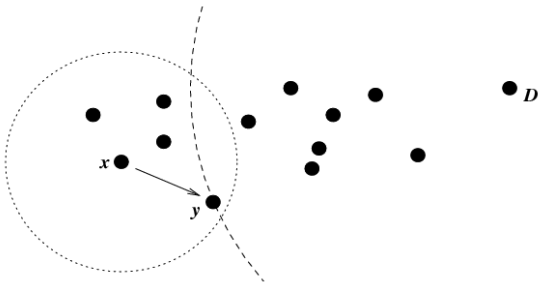


Figure 4: Greedy decision.

(e.g. in presence of obstacles): then the packet must travel around the void region. From greedy forwarding, the packets enter into perimeter forwarding: nodes lying on faces (i.e. edges) closer to the destination are selected as forwarder. The perimeter mode finishes when a node closer to destination is reached (and the greedy mode can be re-entered), or when there are no more faces to traverse. In [10] it is shown that if the source and the destination are in the network, then this solution always work.
Simulations show that GPSR have a high packet delivery success rate, even in presence of mobility (the random waypoint model was used to simulate movement), and that it finds the shortest routes (in terms of number of hops) in almost all the cases, with an overhead sensibly lower than the one produced by DSR.
In [19], the GPSR is extended in order to deliver a packet to the node closest to a given destination.

2.5 Mapping data in locations

As outlined in the previous Sections, sensor networks are so constrained that is not practical to reuse the networking layer proposed for other wireless mobile systems (e.g. ad hoc networks). In general, the networking functionality are to be tuned to the sensor network application, in order to push all the optimizations to the limit. Given the model outlined in Section 2.1, however, it is possible to see the network as a huge set of data producers, and a set of nodes interested in some of the data.

Data centric storage

In order to avoid the huge number of broadcast operations which could arise when trying to localize a data source (or interested destination) if no localization strategy is used, a technique used in the peer to peer community has been borrowed and adapted (see Appendix 4 for a more detailed description of the concept). In [19] is presented a technique called *data centric storage*, based on a geographical routing (a slightly modified version of GPSR [10]).
The whole network is seen as a unique hash table, and every node may use two operations:

- put(key,value), which stores a value identified by key (based on the name of the data),
- get(key), which retrieves the data associated with key.

Note that at the application level it is not necessary (nor useful) to know where the data is actually stored. In practice, a key is mapped on a geographical coordinate using a couple of hash functions. The node that is the nearest to that point of the network is used as storage for the data associated with the key. Given the geographic nature of the routing, it is easy to communicate with the point nearer to a desired coordinate.
Authors present a theoretical performance analysis, based on the following hypotheses:

- a network composed by n nodes, uniformly distributed in an area with high enough density,
- all the paths in the network are composed by $O(\sqrt{n})$ hops, a message flooding costs $O(n)$ messages,
- nodes are programmed to detect some class of events produced by nodes not known in advance, and to take some action after the detection.

It is analyzed the cost of different strategies using as metric the number of total messages sent over the network, and messages processed by the hotspot (i.e. the most intensively used node).
Three strategies to collect data are presented and analyzed:

External storage: Upon the detection of an interesting event, data are sent to an external disk, in which the sink may post queries. The cost (in terms of messages) per event storage is $O(\sqrt{n})$, the cost per sink query is 0 (i.e. it connects just to the disk) and the cost per query posted by nodes is again $O(\sqrt{n})$.

Local storage: Nodes which detect events store locally all the data, incurring in no cost. Queries are flooded to all nodes, with a cost of $O(n)$, and results are returned directly from interested nodes to the sink, at the cost of $O(\sqrt{n})$ per event.

Data centric storage: The technique describe in this Section. After an event observation, data is stored in some node in the network, with a cost of $O(\sqrt{n})$ per event, and queries are directed to nodes storing the right data with the same cost per query.

If D_{TOTAL} is the number of observed events, Q is the number of event types for which queries are issued (assuming only a query per event), and D_q is the number of events detected for each issued query, then it is possible to estimate the following costs:

Total

External Storage: $D_{TOTAL}\sqrt{n}$
Local Storage: $Qn + D_q\sqrt{n}$
Data Centric Storage: $Q\sqrt{n} + D_{TOTAL}\sqrt{n} + Q\sqrt{n}$

Hotspot

External Storage: D_{TOTAL}
Local Storage: $Q + D_q$
Data Centric Storage: $Q + D_q$

Fixing all the terms and letting $n \to \infty$, then local storage will always incur in the highest costs. The ratio of the Data Centric Storage cost over the External Storage cost $(1 + \frac{Q+D_q}{D_{TOTAL}})$ is low if many events are detected.
Thus, data centric storage is the best approach when either n is large or many events are to be detected, but not all will be queried. In the same paper, authors introduce some redundancy in order to provide better fault tolerance while exploiting the geographical nature of the routing: data to be stored by a node is replicated in all of its neighbors[3].
A combination of data centric storage and data diffusion may also be imagined: given a data representation, with the usage of hash function is possible to map it to a node in the network. This node, instead of having a copy of the data, may provide the location of the data source. At this point, the node interested in the data may be inserted in the path with the technique previously described.

Data centric storage oriented to range queries

In [12] a solution called DIMS, oriented to range queries was presented. Range queries are used when data are represented in some ordered space, and users are interested in discovering where (or when) recorded data fell into a subspace. For example, if the sensors record temperature and light conditions, a range query could be the selection of all the events with temperature between $10°C$ and $15°C$, with a light level under $20''$.
Clearly, this problem could be resolved with a simple extension of data centric storage to multiple dimension. When querying for intervals, however, a desired property is that adjacent data are stored in near nodes, otherwise a single query could affect all the nodes in the network many times.
For this reason authors of [12] propose to transform data into geographic coordinates with a locality preserving hashing function, inspired by the $k-d$ trees ([3]).
The basic idea is to assign to every node a unique zone in the network, and to store data in a zone whose

[3]More precisely, in all of the nodes located on its perimeter.

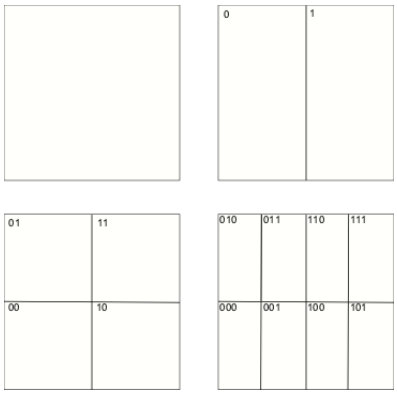


Figure 5: Division in zones after 3 steps.

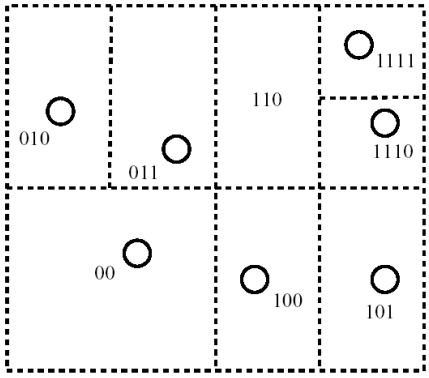


Figure 6: Association of nodes with zones.

code is obtained hashing the data itself. The hashing function however should preserve locality, mapping contiguous data in contiguous zones.

The network is partitioned into hierarchical zones in the following (recursive) way:

First division: the whole network is divided into two zones, labelled 1 (the right part) and 0 (the left one),

i^{th} **division:** if i is even, all the 2^{i-1} zones already existing are divided in 2 equal sub-zone with a horizontal line, each labelled with the same label of the father followed by a 1 (the upper half) and by a 0 (the lower one). If i is odd, then the division is made with a vertical line, and the labels are created by appending a 1 (left part) and a 0 (right part) to the father label.

Figure 5 shows a simple example with 3 steps. Note that the whole division may be represented by a tree, in which internal nodes represents zones divided into sub-zones, leaves are zones not divided, and edges are labelled with 0 or 1. Thus, concatenating labels found on edges connecting the root to a node, the label of the node is found.

The authors propose a mapping between zones and nodes composing a network in which each node controls a different zone. Given that nodes are randomly deployed, the division in zones may not be complete. Every node starts considering all the network as its zone, but with uncertain borders. All the nodes periodically broadcast a beacon containing their position. On the reception of a beacon, a node x can determine if there is another node y owning an adjacent zone, and combining its position with that of the node y, x can determine the new division in zones (a better approximation of real division).
At the end of the process, every node know at least its adjacent zones. The persistence of some undecided border after some time means that the zone in that direction may be empty, or that it is larger than the communication range. Figure 6 shows a simple association. For every zone is defined a backup zone: if the zone is empty, the owner of its backup zone owns it.
When a node generates a tuple of values $(V_1, \ldots, V_m)$ that it must store, it transforms it into a zone code with the simple algorithm in Figure 1 (where normalized values are assumed).
The hashing has the property that tuples with close values (componentwise) are stored in adjacent leaves in the zone tree.

Every node is able to generate the zone in which a tuple of data has to be stored. The routing algorithm, a modified GPSR, is able to correctly deliver the tuple to the chosen zone, or to one which is more suitable for its storage (in case the producer node does not have full knowledge of the division in zones).
The authors also present a method to split queries, in order to directly query the right nodes, and enhance their solution in order to improve fault tolerance (basically by replicating the data destined to a zone Z also in the zone backup(Z).
Performances evaluation is given in terms of sent messages per query (the insertion is exactly the same as in the case of data centric storage, and it is $O(\sqrt{N})$. The complexity depends on the distribution of query ranges, and the authors consider four cases:

- Uniform distribution: a point query is as likely as a whole interval query. In this case the average query complexity is $O(N)$, exactly like when flooding is used.

Algorithm 1 Hashing algorithm

```
Let k be level of node i (i.e. number of bits in its zone code)
Let A_1, ..., A_m be the values to store
dest={}
for i = 0 to k − 1 do
  lb=0;
  bit=1;
  step=i ÷ m;
  for j = 1 to step+1 do
    if A_{i−m·step} ≥ lb and A_{i−m·step} < lb + 0.5/(step + 1) then
      bit=0;
    end if
    lower_bound=lower_bound+0.5/(step);
  end for
  dest=dest.bit;
end for
```

- Bounded uniform distribution: only queries with size bounded by a constant B are possible, and they have the same probability. In this case the complexity is $O(\sqrt{N})$.
- Alebraic distribution: the query range has distribution $p(x) \propto x^{-k}$ for some $1 \leq k \leq 2$. In this case the complexity is $O(N^{2-k})$ (for $k = 1.5$ the complexity is again $O(\sqrt{N})$).
- Exponential distribution: the queries range is distributed following the law $p(x) = ce^{-cx}$. Again, the cost of the query is bounded by the cost to deliver it near the covering zone: $O(\sqrt{N})$.

2.6 Adapting low architectural levels to application needs

As pointed out in this survey, sensors are extremely simple devices which may be demanded to accomplish complex tasks. This is reflected in a software architecture tuned to the application, in order to optimize processing and communication. This approach makes the development of new applications difficult and costly (in terms of time and efforts). For this reason researchers are focusing on general and reusable, yet simple, solutions ad different levels. However, the approach that has been generally adopted can be called a *push* strategy: solutions for a given architectural level are proposed independently from applications, with the goal of optimizing a given metric (e.g. number of exchanged messages, or computational complexity). We think a *pull* approach should be considered: given a (possibly broad) class of application, the whole software architecture should be oriented to it. Low level layers can be highly enhanced by knowing application requirement, without being too specialized (and thus not general enough).
We focus on autonomous sensor network, which do not rely on sinks permanently connected to the network. For this reason we focus on a data centric storage approaches, based on some variant of GPSR routing.
Following the structure of this document, analyzing layer per layer, we make the following considerations:

OS

TinyOS is clearly an indispensable component offering a skinny abstraction of the physical architecture of Motes. As pointed out, it has been voluntarily kept so simple in order to avoid heavy components which could not be useful for all the applications. Its modular design, however, enables the development of reusable components (e.g. MAC layer components, etc). Little work need to be done at this level.

Routing

For the applications we address, we consider routing algorithms based on GPSR, because they are scalable, lightweight and in general well performing. One of the main problems is the location information distribution. In particular, it is highly desirable to achieve accuracy, precision and efficiency. Accuracy and precision is needed when location is an essential information associated to data (e.g. we want to discover where an event

happened). In [4] many proposed solutions are listed with their accuracy and precision (when bounds are given).

Techniques proposed for sensor networks (like the one analyzed in Section 2.4 in general require a all-pairs hop distance computation at least for the border nodes ($O(\sqrt{n})$ of the total network nodes), which is not very efficient. If it is true that this process is run just once in the lifetime of the network, it is also true that this lifetime is, in general, quite limited, and this phase should be optimized.

If the application layer frequently needs multicast or broadcast communication, it should be studied how to efficiently extend the GPSR algorithm.

Middleware

Data centric storage approach made possible to effectively develop sensor networks. However, many problems have not yet addressed. In particular, how to make the system reliable is still an open issue. If data are stored in the network itself before to be acquired by the sink(s), it is very important that they are not lost, even in case of multiple failures. Specially if the stored data are the result of an aggregation process. Analyzed solutions introduce some degree of fault tolerance by means of brutal replication of data in multiple nodes. This solution is obviously less effective (and more costly) than other approaches already known in other fields. Thus, data centric storage should be completed by offering at least reliable (and possibly efficient) storage and communications.

3 Query Processing Issues in Sensor Networks

Sensor networks are deployed to monitor and control the physical environment (sensor field) from remote locations. Most existing architectures are centralized systems where nodes collect data about that environment and send it to an external node (sink) for storage and querying. The development of sensor nodes with some processing power has led researchers to exploit this power to perform simple in-network computations and then transmit processed data to the sink, besides of sending the raw information and leaving to the sink the processing task. Nodes are deployed either inside the phenomenon to be monitored or very close to it.

Sensor nodes periodically sample the surrounding environment producing continuous data streams that grow until nodes run out of battery power. Traditional DBMSs are unsuited to deal with such streams for various reasons:

- sensor nodes produce and deliver data continuously without receiving requests for that data;
- queries over collected data can be less frequent than data insertions;
- produced data has often to be processed in real time because it can represent events, such as traffic accidents, that need a rapid answer;
- queries run continuously because data streams never terminate, so, they can see system conditions change during their execution;
- because of storage constraints, an entire stream can not be stored in the disk;
- because data streams are possibly infinite, only non-blocking operators can be used;
- the rate sensors nodes delivery data at is not reliable because a local source of interference can interrupt wireless sensor connections; so data can be dropped, delayed or garbled; furthermore, if an operator needing a sensor tuple tries to pull it from the sensor, it blocks if the tuple is not available, so, operators must process data only when nodes make it available.

The remainder of the Chapter is organized as follows: Sections 3.1 and 3.2 describe traditional query optimization and issues in data stream management, respectively; Section 3.3 presents some existing approaches to processing queries in wireless sensor networks; finally, we first emphasize differences, positive and negative aspects of these solutions and, then, we outline guidelines for future work in Section 3.4.

3.1 Query Optimization

Given a user query, the DBMS can follow several plans to process it and produce the result [38]. These plans are equivalent with regard to the final result, but can present different costs. Task of the optimizer is to choose the cheapest query plan.

Steps that a query follows through a centralized relational DBMS are as follows:

- the query parser checks if relations and attributes in the query exist in the database; if so, it translates the query into an internal form, usually a relational algebra expression;
- the query optimizer generates algebraic expressions equivalent to the query by varying the order and the implementation of operators and by specifying the use of existing data quick access structures; query plans are usually represented as formulas or in tree form; then, the optimizer compares them on the basis of the cost model and it chooses the one that is estimated to be the cheapest; the cost module specifies the arithmetic formulas that are used to estimate the cost of query plans: it contains a formula for every operator and data access structure;
- the code generator or the interpreter translates the plan produced by the optimizer into calls to the query processor;
- the query processor executes the query.

In the case of flat queries, query tree leaves are relations and non-leaf nodes are algebraic operators like selections, projections and joins. The operator corresponding to an intermediate node is applied to the relations generated by its children and its result is sent up the tree. For complex queries, the number of possible query trees may be enormous. In order to reduce it, the optimizer applies three restrictions in the following order:

- selections and projections are processed on the fly and almost never generate intermediate relations. Selections are processed when relations are accessed for the first time; projections are processed when the results of other operators are produced.
- Cross products are never formed (unless they are asked for by the query) because their results are usually large-sized: relations are always combined through joins. For example, consider a query on the relations R1(A, ...), R2(A,B,...) and R3(B,...) and suppose that it applies the two predicates R1.A=R2.A and R2.B=R3.B; the optimizer generates three plans (modulo join commutativity): (a) it may combine R1 with R2 and the result with R3, (b) it may combine R2 with R3 and the result with R1 or (c) it may combine R1 with R3 and the result with R2; the last plan has a cross product since it joins relations R1 and R2 which are not joined in the query; so, the optimizer produces only the first two plans.
- The inner operand of any join is always a database relation and not the result of another operator because (a) the probability of using indices and hash functions increases and (b) nested joins can be executed in a pipelined fashion.

The restrictions above reduce the number of alternative join trees to $O(2^N)$ for a lot of queries with N relations.

There exist different research strategies that the optimizer may follow in order to find the cheapest execution plan for a given query; for example:

- Dynamic programming algorithm [47]: it builds all alternative join trees that satisfy the three restrictions, by iterating on the number of relations joined so far and it rejects suboptimal trees. The memory requirements and the execution time of this algorithm grow exponentially with the number of joins in the worst case because all partial plans generated in each step must be stored to be used in the next one. Typically, a limit is placed on the number of join (around fifteen joins). For longer queries the optimizer crashes because of memory requirements.

- Randomized algorithms: they address the inability of dynamic programming algorithm to deal with large queries. Most important algorithms of this class think of all the alternative plans for a given query as nodes of a graph: each node has a cost associated with it and the algorithm aims to find the node with the minimum globally cost by performing random walks towards less-cost nodes in the graph.

The optimizer estimates the size of the database relations and of results of (sub)queries and the frequency distribution of attribute values. Several techniques proposed to perform this estimate use histograms. In a histogram on the attribute A of the relation R, the domain of A is partitioned into buckets and a uniform distribution is assumed in each bucket: for any bucket b and for any value $v_i \in b$ the frequency f_i of v_i is approximated by $\sum_{v_j \in b} f_j/|b|$. There exist various classes of histograms:

- trivial histograms: the uniform distribution is made;
- equi-width histograms: the number of distinct attribute values is the same in each bucket;
- equi-depth histograms: the sum of the frequencies of the attribute values is the same in each bucket;
- serial histograms: the frequencies of the attribute values in each bucket are either all greater or less than the frequencies of the attribute values associated with any other bucket; under various optimality criteria these histograms are optimal the worst-case and average error; the problem is that identifying the optimum serial takes exponential time in the number of buckets;
- end-biased histograms: some number of the highest frequencies and some number of the lowest frequencies in an attribute are maintained in separated buckets and the remaining frequencies are put into a single bucket; they are serial histograms, but identifying the optimum end-biased histogram takes only slightly over linear time in the number of buckets; besides, end-biased histograms require little storage since most of the attribute values are in a single bucket and do not have to be stored explicitly; finally, estimate errors are not usually too far off from the corresponding errors that are got with serial histograms.

In parallel databases, besides choosing the implementation of each operator, the use of indices, if and when duplicates must be eliminated, and so on, the optimizer makes two further choices: the number of processors that should be given to each operator (intra-operator parallelism[4]) or placing operators into groups that should be executed simultaneously by the available processors (inter-operator parallelism that can be subdivided into pipeling and independent parallelism). The scheduling alternatives arising from these choices increase further the number of execution plans for a given query. Thus, most systems adopt some heuristics to reduce the plan space. In the two-stage approach [37], in the first phase the optimal sequential plan for a given query is identified using techniques outlined above; then, the optimal parallelization of that plan is found.

In early distributed databases, because the network cost was dominant, semijoins were used in order to only transmit the tuples that would have contributed to join results [27, 39]. Another solution to efficiently execute joins between tables stored on different sites used Bloom filters[5], large bit vectors that approximated join columns and were transferred across sites to determine which tuples might have participated in a join so that only these might be transmitted [39]: suppose that a user issues a query at the site S_1 joining two relations R_1 at the site S_1 and R_2 at the site S_2 and that join attributes are $R_1.A$ and $R_2.A$; a simple query plan could be sending a copy of R_2 from S_2 to S_1 and executing the join over S_1. In the early distributed system this was particularly expansive since bandwidth was a limited resource; bandwidth requirement can be reduced by using Bloom filters: a Bloom filter from column $R_1.A$ is generated and sent to S_2; table R_2 is scanned and the value $R_2.A$ in each tuple is hashed (hash functions h_i used in the site S_2 are the same as the site S_1): if the filter contains the bit 1 at positions $h_i(S_2.A)$, that tuple is send to S_1; here, R_1 and the stream coming from the site S_2 are joined and the result is returned to the user.

[4]Deciding how many processors must be given to each operator is an important issue also in sensor networks where operators can be executed by one or more nodes.

[5]Bloom filters are a technique to control if a given element can or can not belong to a set [28]; even if they return true, there is a non-null probability that researched element is not in the set.

Predicate Migration Algorithm, which produces an optimal query plan for queries with expansive methods, is presented and proved in [36]. Expansive methods are all those predicates the evaluation of which costs more than retrieving data referenced in the predicates. For example, suppose you wish to know in what films your favourite actor played, namely, if there exists at least one scene where that actor appears; thus, your query will contain a predicate which, for each film, searches for actor's image in each scene; the evaluation of such a predicate is not trivial and clearly takes a long time. If the query to be optimized contains no expansive methods, then Predicate Migration Algorithm is circumvented and standard query optimizers are used. Predicate Migration slightly increases the time required to optimize any query: the additional cost factor is polynomial in the number of operators in a query plan.

First, the case of queries over a single table is discussed and it is showed how to order predicates in such a query in order to minimize the cost of applying them to the table. Let us consider the following query where the table R has scheme R(A, ...) and $p, p_1, ..., p_n$ are predicates over R:

```
select R.A
from R
where p and p1 and ... and pn
```

A traditional optimizer orders these predicates arbitrarily and, thus, it may apply expansive predicates before cheapest ones. Let us see how Predicate Migration Algorithm solves the problem: supposing that an index has been defined on one or more attributes of the table (such as p), all the predicates over those attributes are applied first because their cost is almost zero; in fact, they are not actually evaluated but the index is scanned and only the tuples satisfying predicates are retrieved. If p_1,..., p_n are the subsequent non index predicates in the order in which they are applied to each tuple of the base table, and supposing that distinct predicates are independent and any predicate p_i costs c_i and has selectivity s_i, then the cost of applying all the non index predicates to the output of a scan containing N tuples is:

$$c = c_1N + s_1c_2N + ... + s_1s_2...s_{n-1}c_nN$$

Migration Predicate Algorithm orders predicates in ascending order of the metric:

$$rank = \frac{selectivity-1}{cost}$$

Note that since the selectivity of any predicate is from 0 and 1 inclusive, the rank is never positive: the rank of any predicate is low if (a) the cost of the predicate is low and (b) its selectivity is low. Query cost model incorporates both selectivity and cost estimates for predicates; in fact, if predicate costs may differ by several orders of magnitude, evaluating predicates only in increasing order of selectivity is not an efficient strategy because there is the risk to apply first very expansive predicates, whereas evaluating first cheap ones would be better in order to reduce the number of tuples which expansive predicates must be applied to. Note that changing the position of predicates having the same rank has no effect on the cost of the query plan. It was demonstrated that the algorithm is guaranteed to terminate in polynomial time and that it produces the optimal plan tree.

Consider the case of queries with both selections and joins. Given a query join plan, let us see how to minimize this plan's cost (i.e., how to place optimally predicates over a single table and predicates over multiple tables in the join plan) under the constraint that the order of the joins may not be changed. A stream[6] in a plan tree is a path from a leaf node to the root. A plan tree is optimized treating each of its streams individually and sorting the nodes in the streams in ascending order of their rank. If the Predicate Migration Algorithm is applied to all the possible join plans for a query, it is guaranteed to produce as output a minimum cost plan for the query, which is the unique semantically correct and join-order equivalent tree with only well-ordered streams. Unfortunately, Predicate Migration does not get along with traditional query optimizers because they discard the join plans of suboptimal cost in order to reduce space and time required to optimize queries with a lot of joins: the optimum of a stream could be obtained by optimizing a join plan that the traditional optimizer has, however, pruned. But, if a subexpression has no expansive predicates, its plans can be pruned because its predicate places will not be changed by Predicate Migration Algorithm; or, if all the expansive predicates in a subexpression have greater rank than the rank of any join in any plan for the subexpression, then the expansive predicates may be pulled to the top of the plan for the subexpression and the part of the subexpression without the expansive predicates may be pruned as usual.

[6]Please, note that here the word "stream" is used with a different meaning from the rest of Chapter 3.

3.2 Stream Query Processing

A data stream is a real-time, continuous and ordered (by arrival time or a timestamp) sequence of data items [35]. It is impossible to control the arrival order and to locally store an entire stream. Individual data items may be modelled as relational tuples (e.g., [45]) or instantiation of objects (e.g., [29, 49]). Queries over streams run continuously and return new results incrementally as new data arrives. Unless window techniques are adopted in order to extract a finite number of values from a stream, data streams have to be processed on-line and back-tracking over a stream is not feasible because of storage constraints.

3.2.1 Blocking/Non-blocking Operators

Projection and selection operators can be used in queries over streams without any modification compared with traditional database. Other operators, such as sorts, aggregates and a few join implementations, cannot be applied to a data stream because of their blocking nature. Such operators can be redefined to allow aggregates and joins which take streams as input and produce results incrementally: whenever a new tuple arrives, the aggregate is updated and its revised value is delivered to the user. If traditional blocking operators must be used, there exist three techniques for unblocking them:

- asking the user to specify a subset (window) of the stream which they operate over: this subset can be defined by a number of samples or by two time bounds;
- evaluating incrementally operators in order to avoid scanning a window several times (for example, the AVERAGE aggregate may be incrementally updated by maintaining the cumulative sum and item count);
- exploiting stream constraints: schema-level constraints include synchronization among timestamps in multiple streams, clustering and ordering [26, 34, 45]; data-level constraints may consist of control packets inserted into the stream that specify any conditions which will hold for all future data items, for example no other tuple with timestamp smaller than τ will be produced by a given source.

3.2.2 Aggregation

Aggregation enables information about large amounts of data to be summarized; the idea is to represent a set of items by a single value or to classify items into groups and determine one value per group. Typical aggregates are minimum, maximum, sum, count and average. Aggregation can be a useful operation over streams because it reduces the quantity of data to be transmitted and store. Unfortunately, aggregation is blocking, namely, it requires that all input data are consumed before any output can be produced: this fact and storage constraints make unusable traditional aggregate implementations over streams.

Let us see a non-blocking implementation of aggregates. User Defined Aggregates (UDAs) [51, 53] extend SQL aggregate operators. For example, an (blocking) UDA equivalent to the AVG aggregate in SQL is a procedure whose interface specifies input and output parameters (type and name); the procedure consists of a local table containing partial state records (e.g., partial sum and count for AVG aggregate), and three SQL statement blocks INITIALIZE, ITERATE and TERMINATE that may work on the table: the table is allocated just before the INITIALIZE function is executed and deallocated just after the TERMINATE function is completed. Suppose a user query searches for in what departments does the average of employee salaries exceed S; the semantics is as follows: when the first tuple is processed, the table is allocated and the INITIALIZE statement is executed inserting a record into the table; whenever a new tuple arrives, the ITERATE block is executed inserting a new record or updating an existing record into the table (depends on the aggregate); moreover, if the incoming tuple is the last one, the TERMINATE statement is also executed returning the final result of the computation. Non-blocking UDAs support tumbling windows: the TERMINATE statement will appears void and partial results will be returned by the statement ITERATE. A non-blocking hash-based implementation for the calls of the UDAs has been designed for queries containing a GROUP BY clause: the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE statements; the only blocking operations (if any) are placed in the TERMINATE statement and, so, are executed only at the end of the computation.

3.2.3 Optimization Issues

Traditional cost metrics (e.g., how many accesses on the disk must be done to retrieve data) do not apply to continuous queries over infinite data streams. More appropriate cost metrics can be:

- accuracy and delay vs. memory usage;
- output rate [50];
- power consumption [42, 52].

In particular, last metric is advisable in wireless sensor networks since sensor battery capacity is usually limited: minimizing power usage has the great advantage of extending node lifetime and, therefore, delaying human intervention for recharging or replacing batteries (note that a sensor network may be deployed to monitor environments dangerous or difficult to be reached).

In relational DBMSs, all operators are pull-based: an operator requests data from one of its children in the query plan only when needed. On the contrary, operators over streams process data without making a request to the source. Subsection 3.3.2 presents a solution to reconcile pull-based and push-based operators; a similar approach can be found in [25]. Furthermore, stream management systems are required to support historical queries in addition to processing data produced in real time.

Execution of many similar queries together is needed to ensure scalability. Two approaches are proposed:

- similar queries share a plan that produces the union of results of individual queries [31]; a final selection is then applied to this union;
- query predicates are stored in a table and whenever a new tuple arrives, the table is searched to see what queries this tuple satisfies [30, 43].

The first approach can turn out useful in sensor networks: if multiple queries share single operations or portions of processing, these operations or portions are executed only once in order to save power, making available the result to all involved queries.

In relational databases, given a query the optimizer produces all possible plans by reordering joins in the query and, then, it chooses the cheapest plan according to a particular cost metric. Some work in join ordering has been done for data streams. The cost of a plan can change because processing time of an operator or selectivity of a predicate or arrival time of a stream change [24]: query plan is dynamically updated to match current system conditions; this is accomplished by tuple routing policies that attempt to discover what operators must be scheduled first because they are fast and selective [24, 30, 43, 46].

Data streams can arrive from remote source. Distribution Optimization strategies try to reduce communication cost by reordering operators across sites [32, 48] and performing simple query functions, such as filtering and aggregation, locally at a sensor or a network router [33, 41, 44, 52]. Some specific optimization techniques have been proposed for ad hoc wireless sensor networks in order to decrease communication cost, to save node battery capacity and to manage the fact that wireless links have limited bandwidth and are prone to failures. For examples, since links are shared, if a query contains a MIN aggregate, any node does not deliver its own value m if it hears a value of one of its neighbours smaller then m (Subsection 3.3.3). Moreover, the node could broadcast redundant copies of its minimum to all neighbours in order to minimize the chances of packet loss. This technique is unsuitable for other aggregates such as SUM and COUNT, because duplicate values would alter the result. In this case, a sensor may split its own value and send partial records to each of its neighbours; so, if a record is lost, the remainder of the aggregate should be still computed.

3.3 Query Processing in Wireless Sensor Network

A centralized approach where data are extracted from devices in a predefined way and sent to a unique front-end server for storage and querying, is not suitable for query processing in sensor networks for two reasons [29]: (a) access to the network and processing of queries are separated and (b) valuable resources are used to transfer large amounts of raw (and maybe unnecessary) data to the centralized database; in fact, since sensor nodes are usually powered by batteries with a limited capacity, sensor networks must save energy

in order to extend their lifetime, but transmitting all the data to a central node consumes a lot of energy (communication using a wireless medium is more expansive than processing). Since sensor nodes have some computation ability, part of the processing can be moved from the front-end server into the sensor networks, reducing power consumption.

In the following, we present a few alternatives to the centralized approach, in appearing order.

3.3.1 Cougar Approach

In [29] a model for sensor database systems is defined. Queries usually involve a combination of two types of data: stored data and sensor data. The former includes the set of sensors in the network and their attributes such as their location, and it can be represented as relations. The latter is periodically collected by sensors from the physical environment; in particular, it is generated by signal processing functions that take as input physical signals (e.g., temperature, light, sound, pressure and magnetism) sensed by sensors and produce as output measurements of those phenomena. Since queries can require data aggregation over time windows or data correlation in time, each output of a signal processing function has its production time as a timestamp. Sensor data is represented as time series: time is discrete (natural numbers are used as the time series ordering domain); sensor nodes share the same time scale and their clocks are synchronized; all outputs which are generated by signal processing functions during a same time interval, are associated to a position; if a sensor does not produce data during a time interval, a Null record is associated to the position corresponding to this quantum; whenever a signal processing function produces an output, the record series is updated at the position corresponding to the production time.

Queries over a sensor database are long running because sensor networks produce continuous data streams in theory endlessly, in practice until nodes run out of battery power. A query involves stored data (relations) and sensor data (sequences); except a few cases, a relational operator takes as input base relations or the output of another relational operator, while a sequence operator manipulates sequences or outputs of other sequence operators. During the execution of a long running query, relations and sequences might change: a relation changes because of the insertion of a new record or the delete or modification of an existing one; the only event which can change a sequence is a new insertion. Each query defines a persistent view that is maintained during the time interval the query is running, in order to reflect the updates on the sensor database. Persistent views can be maintained incrementally if (a) updates occur in increasing position order and (b) the algebra used to compose queries does not allow sequences to be combined by any relational operators.

In user's point of view, sensors are modelled as Abstract Data Types (ADT): a sensor ADT is defined for all sensors of a same type; the public interface of a sensor ADT consists of the signal processing functions supported by this type of sensor; an ADT object in the database corresponds to a physical sensor in the real world. Signal processing functions are modelled as scalar functions: their outputs are not modelled as sequences but as result of successive executions of a scalar function during the span of a long running query; thus, queries containing time constraints (e.g., aggregates over time windows) are not supported by the Cougar approach.

As regards the internal representation, queries are processed on a centralized database; the query engine includes a mechanism for interacting with sensor nodes where the signal processing functions are executed and the results are sent to the front-end. The first version of Cougar does not consider a long running query as a view: the system only computes the incremental results that could be used to maintain such a view, these results are obtained by executing sensor ADT functions repeatedly and by combining the produced outputs with stored data. A relational operator is used to model the execution of sensor ADT functions; this operator is a variant of a join between the relation containing the sensor ADT attribute and the sensor ADT function in a tabular form. This representation has been called a virtual relation, virtual because it is not defined in the database system. A virtual relation is a tabular representation of a method; a record in a virtual relation contains the input arguments and the output ones of the associated method: if a method M takes n arguments, then its associated virtual relation has a scheme with $n+3$ attributes, where the first one is the unique identifier of a device, attributes from 2 to $n+1$ are the input arguments of M, attribute $n+2$ is the output of M and the last attribute is the output production time. Whenever a signal processing function returns a result, a new record is appended to the associated virtual relation; records are never deleted or updated in a virtual relation. A virtual relational is partitioned across all sensors of the same sensor ADT; thus, a database is internally represented as a distributed database.

3.3.2 Fjord Approach

In [40] an architecture for query processing over sensor data streams is defined. This approach has two advantages: (a) it allows users to issue queries that combine data streams that are continuously pushed into the system by sensor nodes, with standard source data that is saved to disk and is extracted by traditional operators when applications ask for it; (b) it defines power-sensitive operators (called sensor proxies) that serve as mediators between the query processor and physical sensors. The architecture is as follows: users issues queries to the server; the server processes each query, instantiates operators and locates proxies for the sensors that can have data relevant to this query: the proxy accepts and runs queries on behalf on the sensor, sparing the sensor to send data to all interested users; each sensor delivers sensed data to its proxy; data is keyed by sample time and is logically separated into fields; the proxy converts these values in tuples and sends them to the query processor; when the user stops the query, the proxy stops delivering tuples for that query; the proxy continuously manages the sensor even when no query is running; a proxy is typically used for a lot of sensors. Wired Internet connections or high power wireless radio link proxies to the server.

The sensor proxy has other important functions: in order to save energy and to extend the node lifetime,

- it can adjust the sampling rate of sensors depending on the user demand; in particular, if there is no query on a sensor, the proxy can ask this sensor to power off, or if a sensor is sampling at a rate higher than the one necessary to answer user queries, the proxy can reduce sensor rate;
- if current users are only interested in values within some interval, the proxy can ask sensors not to deliver samples outside this interval;
- the proxy can ask sensor to aggregate values in predefined ways before sending them.

Fjords are the other component of this approach. They generalize traditional approaches to query plans, supporting the combination of data streams and disk-saved data: when a new query is posed, the controller running on the server instantiates operators (join, select, project, ...) and connects local operators via queues to other local operators or to remote operators. Each query running on a machine is allocated its own thread and that thread is multiplexed across operators via a scheduler that directs operators to consume inputs and produce outputs if they are not explicitly invoked by their parent in the query plan.

Each operator has a set of input queues and a set of output queues, takes tuples from the input queues in any chosen order and inserts results in some or all the output queues. Each queue routes data from its input operator to its output operator without any transformation on this data. Fjords support both pull queues and push queues: in the former, the input operator puts data into the queue that can be got by the output operator; in the latter, the input operator produces data invoking its transition method when the output operator calls its get method. Push queues are used to make data streams feasible: whenever a sensor tuple arrives, the sensor proxy inserts it into the input queue of the Fjords that use that sensor as source. If a queue fills, the oldest samples are usually discarded first because they represent less the current state of sensors.

Besides of allocating a separated Fjord for each new query, the use of only one Fjord is proposed for all similar queries over the same sensor since queries over streams are only interested in data produced since the moment queries are issued: this sharing is enabled by instantiating stream scan operators with multiple output queues and storing sensor tuples once in the query processor's memory. Single Fjord approach reduces power consumption compared with multi Fjord approach as (a) there is no overhead due to context switch between threads and (b) sensor tuples have to be put in the buffer pool of the only one Fjord.

The Fjord approach is focused on sensor power saving whereas the Cougar one is more interested in modelling sensor streams.

3.3.3 TAG Approach

TAG [41] offers an aggregation distributed service for ad hoc networks of TinyOS motes. Users pose aggregation queries from a powered basestation with abundant resources; each query is routed to all nodes of network that deliver data back to the user through a routing tree rooted at the basestation; as data flows towards the user, every node aggregates data locally produced with data received from other nodes, as specified in the query.

The routing tree is built as follows: the basestation periodically broadcasts into the network a message asking other nodes for organizing into a tree and it specifies its ID (which is unique for each node) and its level (zero) in the message. When the message arrives to a node whose level is not set, that node assigns its level to be the level in the message plus one, decides that the sender of the message is its parent and rebroadcasts the message after inserting its ID and its level. The tree building ends when all nodes have set their level. When a node wants to send a message to the root, it routes the message to its parent, which forwards it to its parent and so on until the message reaches the root. Routing messages are sent periodically in order to adapt the tree to network changes. Each node which does not transmit a reading for a long time, must periodically send a heartbeat to inform its parent and children that it has not failed.

TAG implements any aggregate via three functions: a merging function *f*, an initializer *i* and an evaluator *e*. In general, the function f is defined as follows:

$$< z >= f(< x >, < y >)$$

where <*x*> and <*y*> are multi-valued partial state records computed over one or more sensor values and representing the intermediate state over those values that will be required to compute an aggregate. For example, in the case of AVG aggregate, each partial state record consists of a pair of values: the cumulative sum S and item count C; thus, f is defined as follows:

$$f(< S_1, C_1 >, < S_2, C_2 >) =< S_1 + S_2, C_1 + C_2 >$$

The initializer i specifies how to initialize a state record for a single sensor value; in the case of AVG, for any sensor with value x the initializer returns the tuple <*x,1*>. Finally, the evaluator e takes a partial state record and computes the actual value of the aggregate; in the case of AVG, the evaluator takes as input the tuple <*S,C*> and returns *S/C*. It is worthwhile noting the correspondence between the three functions *i*, *f*, *e* and the three blocks INITIALIZE, ITERATE, TERMINATE described in Subsection 3.2.2.

TAG classifies aggregates according to four properties that are very important to sensor networks:

- duplicate sensitivity: aggregates may or may not be affected by duplicates readings;
- loss tolerance: exemplary aggregates return one or more values representing the set of all values: their behaviour is unpredictable when losses occur; summary aggregates compute some property over all values: the value computed over a subset of data can be a robust approximation of the true aggregate if either the subset is chosen randomly or the correlations in the subset can be accounted for in the approximation logic;
- monotonicity: when two partial state records s_1 and s_2 are combined via *f*, the resulting state record s' has the property that either $\forall s_1, s_2\ e(s') \geq MAX(e(s_1), e(s_2))$ or $\forall s_1, s_2\ e(s') \leq MAX(e(s_1), e(s_2))$: this property may be used to reduce the number of messages in the network, as explained below;
- amount of state required for each partial state record: in distributive aggregates any partial state record is the aggregate for the subset of data over which they are computed and the size of the partial state records is the same as the size of the final aggregate; in algebraic aggregates the partial state records are themselves aggregate for the subsets, but are of constant size; in holistic aggregates no partial aggregation can be made and all the data must be aggregated together by the evaluator; unique aggregates are similar to holistic ones, except that the amount of the state to be propagated is proportional to the number of distinct values in the partition; finally, in content-sensitive aggregates the partial state records are proportional in size to some (statistical) property of data in the partition, for example see equi-width histograms in Section 3.1.

Query execution is a two phase process: a distribution phase where aggregate queries are sent to all nodes down into the tree, and a collection phase where the aggregate values are routed up from children to parents. Time is partitioned into epochs and only samples collected by devices during a same epoch may be aggregated together. When a node p sends a query request down into the network, it specifies a time interval it expects to hear partial state records from its children. When any child, for example q, receives the request, it wakes up, synchronizes its clock with the time information in the message and, then, delivers the request to its children setting the delivery interval for them to be before the time its parent expects q's record. The

distribution phase continues until all the nodes have received the query request. During the epoch after the query propagation, each node listens for messages from children during an interval longer than one specified, in order to overcome synchronization problems. It computes a partial state record combining children's values with its local sensor readings and it transmits this record up the routing tree during the interval requested by its parent.

A query may contain a grouping clause; in this case, the root inserts the grouping condition into the query request to be routed down the network. Query execution follows the approach described above except that partial state records are tagged with a group ID. Any leaf node applies the grouping expression to compute a group ID, tags its record with this group and sends it to its parent. When a node receives a record from a child, it compares its own group ID with the one that is in its child's record: if they coincide, the node combines the two values according to the aggregation function specified into the query; otherwise, the node stores its own value and the received one separately. If another child's message arrives with a value in either group, the node updates the right aggregate. During the next epoch, the node delivers to its parent the values of the groups about which it collected information during the previous epoch. If the groups to be stored exceed the available storage in any node, one or more groups must be evicted.

The main advantage of TAG compared with the centralized aggregation approach is to reduce the communication required to compute an aggregate. Other important benefits are as follows:

- TAG tolerates disconnections and loss: nodes communications can be interrupted by a local interference source, thus, aggregation requests or partial state records can result garbled or updated or lost; furthermore, devices can run out of power. But, since channels are shared, nodes can reconnect by snooping on their neighbours' partial state records if information about queries is included in such records.

- Each mote has to transmit only one message, whereas in the centralized approach nodes at the highest levels have to route more records than the ones at the deepest levels.

- The fact that time is divided in epochs, enables to know when the processor is idle: during these intervals, the radio and the processor can power off in order to consume very little power. TAG provides for a periodic bootstrapping phase where nodes (in sleep mode) can wake up, learn queries currently in the system, know its parent and synchronize with neighbours: devices which have not heard from their neighbours, listen for several times during this period between sleep intervals.

In [41] there are presented several optimizations using the fact that motes communicate over shared channels:

- snooping can be used to reduce the number of messages sent for monotonic exemplary aggregates (e.g., MIN and MAX): if the query contains a MIN operator, a mote does not send its own value if it has heard a neighbour transmitting a value less than its one;

- in an alternative technique for computing monotonic exemplary aggregates, such as a MIN, the root computes the MIN on the highest k levels, then it aborts the result M and sends a new request asking for values less than M: leaf nodes do not deliver a partial state record if their values are greater than M, while intermediate nodes, however, must still forward records;

- network faults are monitored and adapted to at two levels: first, each node maintains a list of neighbours and monitors the quality of the link to each of its neighbours by tracking the packets received from each neighbour; each node assigns a locally unique sequence number with each message; if a node n observes the link quality to its parent p is worse than that of some other node q, it chooses q as its parent if q is closer or as close to the root as p and the node n is not q's parent. Second, if a node n has not heard from its parent for some fixed period of time, it assumes that its parent is failed or moved away; thus, it chooses a new parent from its neighbour list; if n chooses as its parent a node of its subtree, its children must reselect their parent.

3.3.4 TinyBD Approach

TinyDB [42] is a distributed acquisitional query processor that runs on each node in a sensor network. TinyDB runs on Berkeley Mica motes on top of the TinyOS. It represents the current state of the art. It

uses the ability of smart sensors to control where, when and how often data is acquired in order to reduce power consumption; furthermore, it has a lot of features of a traditional query processor, such as selecting, projecting, joining and aggregating data.

TinyDB assumes the existence of a unique table "sensors" with a column for each different type of physical sensor in the network; at regular sample intervals a new tuple is appended to the table for each sensor node, containing the unique ID and the location of the node, the sampling time and the reading of each sensor physical.

The basic architecture is as follows: users submit their queries to a basestation (i.e., a powered PC) where they are parsed and optimized to choose the order of sampling, selections and joins on individual nodes that minimizes power consumption (note that sampling data and transmitting results cost more than running single operators). Each node maintains a catalog of metadata associated with its attributes, events and user-defined predicates: each event has a name, a signature and a frequency estimate; user predicates have a name, a signature and a selectivity estimate; finally, metadata associated with each node attribute are: how many 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: TinyDB adopts the implementation of aggregates via three functions that initialize, merge and update the final value of partial state records as they flow through the system (see Subsection 3.3.3 above). Finally, TinyDB stores the cost of processing and delivering sensed data. Each node periodically sends a copy of its own catalog to the basestation where it is used by optimizer.

Sampling costs more than executing an operator, but if a predicate is required to be evaluated over an attribute of a sensor node, the corresponding sensor must sample the physical environment. However, if a predicate discards a tuple of the table "sensors", all subsequent predicates must not examine that tuple and, so, the expense of sampling any attribute referenced in those predicates can be avoided. Thus, these predicates are expansive and must be ordered carefully. The problem of ordering expansive predicates has been solved in traditional databases by Predicate Migration Algorithm, described in Section 3.1, but it is somewhat different in TinyDB for two reasons: (a) expansive predicates are on a single table ("sensors") and (b) an attribute can be referred in multiple predicates. To overcome the problem, TinyDB treats the sampling of a sensor as a particular operator and it tackles the issue 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) besides of assuming a priori existence of data, a data is 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.

The second important aspect of TinyDB optimizer is about event-based queries, namely, queries that are instantiated and executed only when particular events occur. It is possible for multiple instances of such a query to be running at the same time, but this presents a problem: each instance of the query consumes significant energy sampling and delivering results, apart from the other instances. In order to reduce power consumption, a multi-query optimization technique based on rewriting is adopted: any event-based query is rewritten as a sliding window join between the stream of events and the table "sensors". This techniques works badly if event frequency is low because the overhead of checking if event queue is empty is greater than the advantage of having only one query running.

After a query has been optimized, the basestation broadcasts it into the network. When a node receives the query, it controls if it and/or a few of its children in the routing tree can produce data for the query; in the negative case, the subtree rooted at that node may be excluded from the query saving the energy to disseminate, execute the query and transmit its results on several nodes. Suppose that what nodes have to participate in the execution of queries, is indicated by a selection predicate on a constant attribute; to enable each node to decide if a query applies locally and/or to its children, a semantic routing tree (SRT) is used. Building an SRT is a two phase process: first an SRT building request is flooded by the basestation into the network; this request specifies the attribute A the tree will be built over. When a node p receives the request, if it has no child, p chooses a parent q from available ones and sends to q a parent selection message that reports p's value of A and p's ID. If p has children, it delivers the SRT building request to all of its children; when p has heard from all of its children, it chooses a parent and sends to it a selection message indicating the range of values of A which it and its children cover. The first phase continues until all nodes have heard the request. After the optimization, a query with a predicate over A is forwarded by

the root down the SRT and when it arrives at a node p, p checks if its interval of values of A has non null intersection with the interval of values of A satisfying the query predicate. If so, p forwards the query to all of its children and it prepares to receive their results. Also, if the query applies locally, p starts executing it. If the query applies neither to p nor to any of its children, p forgets the query.

Once a query has been disseminated into the sensor network, the query processor begins executing it. Time is divided in epochs. Nodes sleep for most of an epoch to minimize power consumption. Furthermore, nodes are synchronized: they sleep in the same time intervals and they wake up at the same instants in order to avoid losing results because a child sends a message to its parent while it is sleeping. When a node wakes up, it samples sensors at the rate indicated (and dynamically adjusted) by optimizer based on lifetime estimates and information about radio contention and available power. Samples are filtered and routed to join and aggregation operators as specified in the query plan. Results are put into a queue with data received from children, waiting for delivery to the node parent. The queue can fill, depending on the cardinality of join, the number of queries running and the number of aggregates and groups. In that case, the system must decide how to manage this situation: it can retransmit this value or combine it with some other value for the same query or discard the tuple.

Note that TinyDB extends the TAG approach to cover all of types of queries and, in the mean time, it improves TAG because every new query is only sent to nodes that can have data useful to answer that query, instead of flooding the query to all the nodes.

3.4 Discussion

Let us compare the approaches outlined in Section 3.3, focusing on their differences, strength and weakness. The four solutions belong to two different categories: Cougar proposal and Fjording architecture are enhanced centralized architecture, whereas TAG and TinyDB are first attempts at a distributed execution of queries in sensor networks. The first two solutions improve the centralized architecture by extracting from the network only data required by user queries instead of all the data acquired by sensors from the physical environment; the reason of this is to reduce power consumption on network nodes since sensor node battery has limited capacity. But, as in the traditional centralized approach, data is still stored in a database on a fixed node and retrieved from it in order to answer user queries. The two approaches focus on different and complementary aspects of the problem: Cougar proposes a model for data streams and long-running queries on such streams by using abstract data types. Cougar does not consider energy constraints of sensors because it is devoted to battery powered sensors and it does not address the push-based nature of sensor streams. On the contrary, Fjording approach adopts Cougar's data model, but it focuses more on the issue of energy-efficiently processing queries over never ending streams: first, it provides power-sensitive operators; second, it supports shared operators for similar queries over the same sensor; third, proxies change sensors' sampling and transmission rate according to user demand.

TAG is a first attempt to reduce power consumption on sensor nodes required to delivery data to a central server: it uses the limited processing power of sensor nodes and takes part of the processing from the server into the network. But it has two limits: (a) it deals with only aggregation queries and (b) queries are sent to all nodes, also to nodes that have not data useful, which wastes energy and bandwidth. However, it is worthwhile remembering that TAG presents several advantages compared with executing aggregate queries in a centralized architecture: (a) in-network aggregation decreases traffic (and so, energy and bandwidth requirement) through nodes; (b) transmission load is the same on any node no matter what their level is in the routing tree, thus, power consumption is fairly uniform on each node; (c) it is a loss tolerant, power sensitive and topology change adaptive solution and (d) it makes it easy to identify intervals where the processor and the radio may be turned off over each sensor node.

TinyDB overcomes both limits of TAG by building an index over each constant attribute and using such an index to send a query to only those nodes that can participate in query execution. TAG and TinyDB address the issue of having a power sensitive sensor network, but from different points of view: the former stresses the importance of in-network query processing, while the latter believes the target can be pursued by controlling when and how often environment must be sampled and it introduces event and lifetime clauses for this reason: event-based queries are instantiated and executed when another query or the operating system generates some given event; lifetime-based queries must be executed for a given time interval (specified by the user in the query) and physical environment is sampled at a rate as high as possible by taking available

node power and the lifetime requirement into account.

To our knowledge, physical and logic levels coincide in both TinyDB and all the other existing solutions. This presents a few problems that have induced us to consider it useful to separate them. Let us investigate these issues identifying several guidelines for future research. In TAG and in TinyDB there exists identity between routing at network level and query routing, but this implies that any change of network topology modifies query execution strategy.

Both TAG and TinyDB suppose that messages are exchanged by nodes using a routing tree and that when a node sends a message to another node, the latter receives it. However, they do not address the issue of what doing if data gets lost because the receiver is not hearing for any reason (interference, lack of energy, ...). Some mechanism of buffering could be introduced to save data whenever the receiver is disconnected, so that it can receive when it connects again. Buffering is an expansive operation, given storage and energy constraints over sensor nodes.

Let us see where different operators are executed. In general any aggregate is executed on the node generating data (see Subsection 3.3.3 above): each node combines locally produced data with data received from neighbours. For other operators, the routing tree or the SRT guides the choice of processing site. However, it would be more efficient having distributed processing techniques based on load balancing, energy saving, data reuse. Let us examine the first aspect: if a node must process a lot of queries, its battery capacity runs out faster than other nodes: that node can delegate part of its task to another node. About the second issue, if one query shares some or all the operations with other queries, as Fjording architecture suggests, these operations can be executed only once at the most suitable site, saving energy.

An inefficient aspect of current approaches is the existence of a sink node linked to the node a user issues queries from (i.e., the root of the routing tree or the SRT). That presents two problems: (a) if the sink is a mobile device, when it moves, the routing tree can change; (b) the root runs out faster than other tree nodes, so that the sink becomes disconnected from the sensor network.

Cougar, TAG and TinyDB view sensor data as a single table "sensors" with a column for each different type of physical sensor in the network, which a new tuple is added to for each sensor node at regular intervals; organizing acquired data into only one table has two main disadvantages: (a) each node is not said it has all possible physical sensors and, if so, all of its tuples will present a Null value at the positions corresponding to missing sensors, thus, the table "sensors" can contain rows with a lot of Null values: memory is wasted; (b) moreover, think of a system where more frequent queries are not "what is the average temperature in the rooms on the first floor at CNR?" but "what is the average temperature near the window (or at the ceiling or at the floor or on the walls) in the rooms on the first floor at CNR?"; in this case, the table "sensors" would not contain only one column for temperature sensor but one column for each logical temperature sensor, so that Null values in the table and the waste of memory would increase; maybe, it would be worth creating a table grouping logically related sensors, which defines what sensors sensing temperature at the window are, and so on.

4 Conclusions and future work

This deliverable presents the state of the art in query optimization and processing over data produced by sensor networks, considering current networking and architectural issues in such a class of systems.
In this respect we assume that the underlying sensor network architecture complies to the following characteristics: the network is wireless and multihop, and it is composed by heterogeneous sensors (with respect to computational, storage, sensing and energy capacity). The network is accessed through one or more special mobile nodes, called sinks, which are not permanently connected: they connect to the network to issue new queries and/or to collect results of previous queries.
We organized this review to reflect the classical (yet minimal) layer division in the software architecture. It should be observed, however, that in the current solutions, which are either pushed from low layers (e.g. the data centric approach, Section 2.5, offering a get/put repository to higher levels) or completely implemented at higher layers (e.g. TinyDB - Section 3.3.4), the layers we identified tend to collapse due to the complexity of developing software for devices like MOTES (Section 2.2).
We believe that a soft abstraction could greatly improve the expressive power of databases processing streams of data produced by sensors, while not demanding much more in terms of nodes performances. A better distinction between functionalities could lead to improvements in saved energy, performance and expressive-

ness.
For example, TinyDB is constrained to use a poor routing (available as a module running on top of the TinyOS), and in order to disseminate queries and collect the results a semantic routing tree must be built. If the network topology changes, the routing tree must be reconstructed, with a waste of energy. Using a more flexible routing (like GPSR, Section 2.4) and defining query processing strategies independent of routing, complex query plans and distributed query processing can be more effectively employed.
In TinyDB, the query optimization is executed by the sink node, which chooses a query plan that minimizes a given cost measure (Section 3.3.4). However, energy is explicitly taken into account only when deciding the sampling rate (a user can request a minimum network lifetime), while the sampling and pruning order is decided based on the cost of sampling attributes (and only in the case of static or slowly varying attributes). We believe that with more information coming from lower levels (and with an efficient way to share and access them - e.g. a data centric storage approach, Section 2.5) the query optimization may be enhanced, e.g. considering the sampling cost as increasing with energy consumption. Balancing the energy consumption among all the nodes could result in a longer network lifetime without affecting the sampling rate.
This aspect is even more dramatic in heterogeneous networks, where lower capacity nodes may delegate part of their tasks to other, more powerful nodes.
The commonly adopted approach considers sensors as sources of data streams, and the network as a unique table, composed of a set of columns, one for each sensor type. Such a solution is not flexible from the user point of view: theese concepts can be generalized, offering the opportunity to create logical table/streams aggregating simple ones, and possibly offering a wide choice of compositional methods. In order to enable such an enhancement, a low level *stream system* (the assonance with file system is not accidental) should be developed, offering basic functionalities over simple streams, like creation, opening and input/output. At an higher level, such a service could be used to aggregate streams into new (logical) streams, offering to the database (and to the user) a powerful mechanism to manage and control sensed data.
The above requirements may impact on the lower layers, which should be reconsidered under this aspect. In particular, unicast routing, data centric strategies and reliable and secure data storage and communication services are expected to be the key elements to support this framework.

Appendix A

Peer to peer strategies for service localization

The peer to peer approach to distributed software is being adopted in an increasi ng number of cases, pushed by the success of many file sharing applications and by the advantages such an architecture offers. One of the drawbacks of the first products was the data localization: a query had to be sent to all the nodes composing the network, in order to find who could serve it. This causes a huge number of communications in the network, and a heavy usage of every node, which must serve all the queries for all the data.
We present here a few interesting solutions to this problem: CAN ([17]), Pastry ([16]) and Tapestry ([23]), Chord ([21]) and finally a solution based on attenuated Bloom filters ([18]).
All but the last approach may be seen as a mapping of nodes and queries onto some topology in a d-dimensional space. The mapping allows for a quick location of data, to improve routing. The solution based on attenuate Bloom filters is a randomized approximation: data stored in some nearby node can be obtained quickly with high probability. Otherwise, a deterministic solution is used.
We do not discuss in great detail all the issues every mechanism must consider (e.g. how to react to node failures, etc), limiting the survey to the basic idea underlying each of them.

CAN ([17])

A Content Addressable Network is exactly a data centric network: data is addressed, not nodes. Nodes are seen as disposed in a d-dimensional torus, and each node is responsible for a subspace disjoint from other nodes subspaces (i.e. the torus is partitioned).

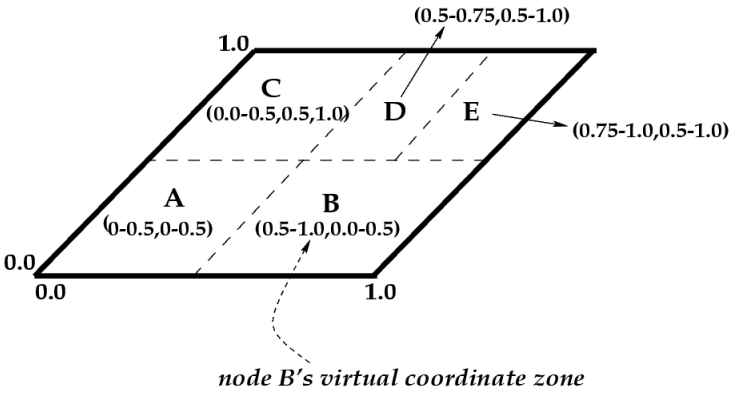


Figure 7: A content addressable network

Figure 7 (borrowed from [17]) shows a case in 2 dimensions (without the wrapping of sides).
When a new node connects to the network, it requests for an existing node subspace. The node receiving the request gives a part of its space to the newcomer, which enters in the torus. In Figure 7, for example, node A is associated with the square $(0, 0) - (0.5, 0.5)$.
Every node knows its neighbors and is able to determine when one of them leaves. In case of departure of a node, one of its neighbors merges the abandoned area with its own.
Thus, data are mapped in points, and are be stored in/retrieved from the node which owns the area containing their coordinate. The routing in such a topology is a greedy coordinate-wise approaching, keeping this phase very simple.

Pastry ([16]) and Tapestry([23])

We present these two solutions together, because they are very similar in spirit (i.e. Tapestry has a location management system which is based on a routing layer very similar to the one offered with Pastry).

The topology created may be thought as a generalization of hypercubes: every node has a unique ID represented as a number between 0 and B^a, for some base $B = 2^b$. Thus, if a node keeps a table with $\lceil \log_a(N) \rceil$ rows, with at most B entries per row, it is able to find a path for a given ID with a simple table lookup. In fact, table row i contains a set of pointers to nodes with an ID having a common prefix long i digits. Column j in such a row contains a node with an ID equal to j at the position $i + 1$. Thus, if the destinaion node has an ID with a prefix sharing x digits with the ID of the node that has to find the next hop, and the first different digit is y. The optimal route for the message passes through the node pointed out in row x, column y.

The Pastry solution is an approximation of the optimal approach, which keeps routing tables smaller while offering better fault tolerance (at the expense of slightly longer paths). In practice, not all the possible entries are kept, and in case a needed entry is missing, the packet is routed through the nearest[7] node with the same common prefix of the routing node.

Chord ([21])

The Chord approach can be seen as a monodimensional approach using the technique of consistent hashing ([11]).
Again, every node has a unique identifier (e.g. its IP address) which is reduced to an ID in $0 \dots 2^m$ by means of an hash function. Information about a key k is stored in the node which has an ID equal to the hashing of k (in the first existing successor, if the node is not in the network). Figure 8 (borrowed from [21]) shows a simple case with a network composed by 4 nodes.
If every node is aware of its successor in the circle, then a route from node i to node j may be easily found.

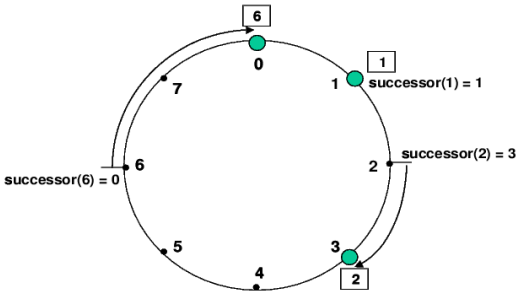


Figure 8: A chord ring.

However, in order to cut the length of the routes, while keeping a small routing table, node i records also the first successor of node $(i + 2^j) \bmod m$. Routing corresponds to tracing a logarithmic number of chords per node in the network, which allow to reach any node in a logarithmic number of steps (if successors information is kept updated).
When a node leaves the network, all the keys it stores are left to its successor, while, when a new node appears, it retrieves the keys to store from its (new) successor.

A probabilistic approach ([18])

This method is quite different from the previously analyzed ones, but it still presents the core components of each:

- compact representation of keys by means of hash functions

[7] The metric used to estimate proximity can be changed following the application needs

- topology knowledge

A Bloom filter is a simple lossy set representation data structure: an array of w bits. The elements of the set are hashed to a tuple of values between 0 and $w-1$, and the bits representing their encoded representation are set. When looking for an element in a set, it is enough to look at the bloom filter elements corresponding to its hashed value. If any of them is set to 0, then the element is not in the set. If they are all set to 1, then there is a certain probability (which depends on the number and quality of hash functions and on w) that the element is present in the set.
It is possible to use the same the same idea to quickly find values in a peer to peer network. Every node maintains a set of bloom filters for every neighbor (where neighbors are nodes nearby following some metric). The i^th Bloom filter associated with neighbor j is the Bloom filter representing the set of keys reachable with i hops through node j. In this way, a node looking for key k asks neighbors in which all the bits representing k are set to 1 at a smaller value.
If the key is within a small number of hops, then it is efficiently found, otherwise a deterministic approach must be used.
An efficient mechanism for disseminate keys to neighbors (in order to keep the Bloom filters updated) is presented in [18]).

References

[1] Ian F. Akyildiz, WellJan Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, pages 102–114, Aug. 2002.

[2] I.F. Akyyildiz, Y. Sankarasubramaniam W. Su, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, (38):393–422, 2002.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–66, August 2001.

[5] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissefinination in wireless sensor networks. In *Proc. ACM MobiCom '99*, pages 174–185, 1999.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[7] D. Johnson, D. Maltz, and J. Broch. Dsr the dynamic source routing protocol for multihop wireless ad hoc networks, 2001.

[8] G.N.C. Kirby, A. Dearle, R. Morrison, M. Dunlop, R.C.H. Connor, and P. Nixon. Active architecture for pervasive contextual services. In *International Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2003), ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Rio de Janeiro, Brazil*, pages 21–28, 2003.

[9] Crossbow Technology. MICA wireless measurement system datasheet. www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA.pdf

[10] B. Karp and H. T. Kung. Greedy perimeter stateless routing (gpsr) for wireless networks. In *Proc. Sixth Annual ACM/IEEE International Confer-ence on Mobile Computing and Networking (MobiCom)*, pages 243–254, 2000.

[11] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[12] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 63–75. ACM Press, 2003.

[13] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003.

[14] The nesC home page: http://nescc.sourceforge.net/.

[15] C. Perkins. Ad hoc on demand distance vector (aodv) routing, 1997.

[16] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[17] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[18] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *Proceedings of INFOCOM 2002*, 2002.

[19] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob. Netw. Appl.*, 8(4):427–442, 2003.

[20] Ananth Rao, Christos Papadimitriou, Scott Shenker, and Ion Stoica. Geographic routing without location information. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 96–108. ACM Press, 2003.

[21] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, pages 149–160, August 2001.

[22] Scott Shenker, Sylvia Ratnasamy, Brad Karp, Ramesh Govindan, and Deborah Estrin. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev.*, 33(1):137–142, 2003.

[23] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[24] R. Avnur and J. M. Hellerstein. *Eddies: Continuously Adaptive Query Processing.* SIGMOD, 2000.

[25] B. Babcock, S. Babu, M. Datar, R. Motwani and J. Widom. *Models and Issues in Data Stream Systems.* In Proc. ACM Symposium on Principles of Database Systems, 2002.

[26] S. Babu and J. Widom. *Exploiting k-Constraints to reduce Memory Overhead in Continuous Queries over Data Streams.* Technical report, Nov. 2002.

[27] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie. *Query Processing in a System for Distributed Databases (SDD-1).* In ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981.

[28] B. Bloom. *Space/time trade-offs in hash coding with allowable errors.* Communications of the ACM, Vol. 13, No. 7, July 1970.

[29] P. Bonnet, J. Gehrke and P. Seshadri. *Towards Sensor Database Systems.* In 2nd International Conference on Mobile Data Management, Hong Kong, January 2001.

[30] S. Chandrasekaran and M. J. Franklin. *Streaming Queries over Streaming Data.* VLDB, 2002.

[31] J. Chen, D. J. DeWitt, F. Tian and Y. Wang. *NiagaraCQ: A Scalable Continuous Query System for Internet Databases.* SIGMOD, 2000.

[32] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing and S. B. Zdonik. *Scalable Distributed Stream Processing.* In Proc. Conference on Innovative Data Syst. Res, 2003.

[33] C. D. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, O. Spatscheck. *Gigascope: high performance network monitoring with an SQL interface.* SIGMOD, 2002.

[34] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan and M. Strauss. *Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries.* VLDB, 2001.

[35] L. Golab and M. T. Özsu. *Issues in Data Stream Management.* SIGMOD Record, Vol. 32, No. 2, June 2003.

[36] J. M. Hellerstein. *Optimization Techniques for Queries with Expensive Methods.* In ACM Transactions on Database Systems, Vol. 23, No. 2, June 1998.

[37] W. Hong and M. Stonebraker. *Optimization of Parallel Query Execution Plans in XPRS.* In Proc. 1st. International PDIS Conference, Miami, FL, December 1991.

[38] Y. E. Ioannidis. *Query Optimization.* Handbook for Computer Science, Ch. 45, pp 1038-1057, CRC Press, Boca Raton, FL, 1996.

[39] L. F. Mackert and G. M. Lohman. *R validation and performance evaluation for distributed queries.* VLDB, 1986.

[40] S. Madden and M. J. Franklin. *Fjording the Stream: An Architecture for Queries over Streaming Sensor Data.* In 18th International Conference on Data Engineering, 2002.

[41] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. *TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks.* In 5th Annual Symposium on Operating Systems Design and Implementation (OSDI), 2002.

[42] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. *The Design of an Acquisitional Query Processor For Sensor Networks.* SIGMOD, 2003.

[43] S. Madden, M. A. Shah, J. M. Hellerstein and V. Raman. *Continuously Adaptive Continuous Queries Over Streams.* SIGMOD, 2002.

[44] S. Madden, R. Szewczyk, M. J. Franklin, D. E. Culler. *Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks.* In Proc. IEEE Workshop on Mobile Computing Systems and Applications, 2002.

[45] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein and R. Varma. *Query Processing, Approximation, and Resource Management in a Data Stream Management System.* In Proc. Conference on Innovative Data Syst. Res, 2003.

[46] V. Raman, A. Deshpande and J. M. Hellerstein. *Using State Modules for Adaptive Query Processing.* In Proc. International Conference on Data Engineering, 2003.

[47] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. *Access Path Selection in a Relational Database Management System.* In Proc. ACM-SIGMOD Conference on the Management of Data, Boston, MA, June 1979.

[48] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin. *Flux: An Adaptive Partitioning Operator for Continuous Query Systems.* In International Conference on Data Engineering, 2003.

[49] M. Sullivan and A. Heybey. *Tribeca: A System for Managing Large Databases of Network Traffic.* In Proc. USENIX Annual Technical Conference, 1998.

[50] S. Viglas and J. F. Naughton. *Rate-based Query Optimization for Streaming Information Sources.* SIGMOD 2002.

[51] H. Wang, C. Zaniolo, C. R. Luo. *ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams.* VLDB 2003.

[52] Y. Yao and J. Gehrke. *Query Processing in Sensor Networks.* In Proc. Conference on Innovative Data Syst. Res, 2003.

[53] C. Zaniolo, C. R. Luo, Y. Law and H. Wang. *Incompleteness of Database Languages for Data Streams and Data Mining: the Problem and the Cure.* SEBD, 2003.