ADAGA – ADaptive AGgregation Algorithm for sensor networks

Angelo Brayner[1], Aretusa Lopes[1], Diorgens Meira[1], Ricardo Vasconcelos[1], Ronaldo Menezes[2]

[1]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR), Brazil
[2]Department of Computer Sciences – Florida Institute of Technology, USA

{brayner,aretusa,diorgens,rivo}@unifor.br, rmenezes@cs.fit.edu

Abstract. *Algorithms for query processing in Wireless Sensor Networks (WSNs) should be able to handle resource limitations such as memory and battery life. Adaptability has been explored as an alternative approach when dealing with these conditions. Adaptive algorithms can adjust their behavior in response to specific events that take place during data processing. In this paper, we propose an adaptive algorithm for processing in-network aggregation in sensor nodes of a WSN, called ADAGA (ADaptive AGgregation Algorithm for sensor networks). The ADAGA adapts its behavior according to memory and energy usage by dynamically adjusting data-collection and data-sending time intervals. The results obtained through experiments prove the efficiency of ADAGA.*

1 Introduction

Sensors are devices used to collect data from the environment to detect or measure physical phenomena. Sensors are limited in power, computational capacity, and memory. In general, wireless sensor networks (WSNs) consist of groups of sensors where each group is responsible for providing information about one or more physical phenomena (e.g., group for collecting temperature data). Such groups use a WSN to disseminate the data collected in certain geographic regions. WSNs are mainly characterized by: *(i)* having a large number of sensor nodes; *(ii)* generally using broadcast communication; and *(iii)* having their location frequently changed [1].

A common organizational structure for WSNs consists of groups of sensors sending data to a sink node or base station that has robust disk storage, no energy restrictions, and capacity of processing [15]. We consider a network scenario where a base station receives data from groups of sensors scattered in the network. Sensors and base stations are organized as a scale-free network [3], which has a base station as the hub node of a set of sensors. Base stations are aware of node hierarchy in the network – the information about the hierarchy is updated regularly in order to reflect possible changes in node locations. In this model, each sensor has the capacity to aggregate local and incoming data (from other nodes), in such a way that data packets are passed from sensor to sensor until they reach a base station.

WSN applications frequently work with queries executed over continuous data streams [6]. For example, a sensor used to collect temperature could be configured to continuously get information from the environment. In this case, the amount of collected data may become very large. Some approaches deal with data streams by limiting the amount of data received. Considering common available bandwidths, large data volumes may produce heavy traffic congestion in the network. An approach to solve this problem is to aggregate data before sending them, thus reducing the amount of traffic in the network. An extension of this approach consists in aggregating data progressively as data are passed though the network – this is called in-network aggregation [4][12][14]. This technique reduces the amount of data to be transmitted by sensors and consequently the data traffic in the network.

Query-processing algorithms for WSNs should be able to handle conditions such as failures, resource limitations, the existence of large amounts of data streams, and mobility of the sensors, which are all common characteristics of WSNs. Adaptability has been explored as an alternative to deal with these conditions. Adaptive algorithms can adjust their behavior in response to specific events happening during data processing. In this paper, we propose an adaptive algorithm, called ADAGA (ADaptive AGgregation Algorithm for sensor networks), for processing in-network aggregation in WSNs. The ADAGA adapts its behavior according to memory and energy usage by dynamically adjusting data-collection and data-sending time intervals.

We also describe a generic data model for WSNs, which allows a logical view over data streams handled by the system; and a SQL-like query language, denoted SNQL (Sensor Network Query Language), which provides the necessary support for the specification of declarative queries for WSNs. We argue that SNQL gives more flexibility to applications because: (*i*) users can control the data volume and the precision of the results by defining parameterized features (clauses) in advance; (*ii*) users are allowed to change clause values of a query on-the-fly; (*iii*) the language provides support to continuous queries; and (*iv*) there is support to predefined queries (injection of the same query into the system an indefinite number of times – each time a new query instance is performed).

This paper is organized as follows. Section 2 briefly describes our considered model of a wireless sensor network. Section 3 describes our data model and specifies the proposed query language, SNQL. In Section 4, the ADAGA algorithm for processing data in sensor nodes is presented and evaluated. Section 5 discusses the related work and Section 6 concludes the paper.

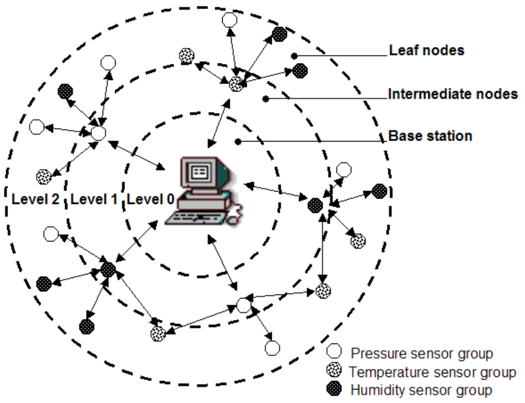
2 Wireless Sensor Network Scenario

In general, sensors are battery-powered devices applied in monitoring applications used to detect specific events (e.g., an animal movement) or collect information about some environmental properties (e.g., pollution levels). They are particularly important for applications with little or no human interventions (e.g., monitoring deep oceans currents). Some premises are usually admitted for sensors [1]: *(i)* they have limited energy, computation and storage capabilities; *(ii)* they have a simple architecture consisting of a sensing unit, a processing unit, a transceiver unit, and a power unit; *(iii)* they collect only one or very few types of data; *(iv)* they are prone to failures, but these failures should not strongly affect application results; and *(v)* they usually broadcast data by means of radio transmissions.

It is important to note that the sensor node lifetime is extremely dependent on the available energy in its battery. There are three domains to be considered in energy consumption [1]: *(i)* sensing activity (data collection from the environment), which is the primary goal of a sensor; *(ii)* communication (sending and receiving packets), which is essential to form a WSN; and *(iii)* data processing, which consists in some operations applied over data by smart sensors [5][13]. Even though all these activities waste energy, communication is responsible for the bulk of the power consumption hence being the main point of attention in algorithms designed for sensors – save energy by reducing the communication activity [13], which consequently increases WSN lifetime.

A common architecture proposed for WSNs is based on the distribution of sensor nodes in a geographical area in such a way that sensors send collected data to a base station (a hub) by using multi-hop routing protocols. Usually, these approaches organize sensors in routing trees [11][12][14].

In this paper we assume a scale-free network organization, in which sensors are categorized by their sensing activities (e.g., temperature collection). A set of sensors with the same sensing activity forms a sensor group. It is important to note that sensor nodes that have *N* different sensing activities will be categorized in *N* different sensor groups. For simplicity, we consider a WSN that consists of just one base station and several sensor nodes. The generalization of this architecture would be to link base stations by transmission media in order to guarantee network scalability. Nevertheless, there is a base station, in which a query is injected into the system and to where results are generated. A simplified scale-free network is organized as described in Figure 1.



The three types of nodes:

Base station: it is a robust node (in Level 0) that organizes the network topology, distributes queries to sensors, receives data sent by sensors and returns results to users.

Intermediate nodes: they are distributed in a geographical area in intermediate levels of the network (e.g. Level 1). These nodes collect data from the environment, receive data packets from other nodes (parent nodes), process in-network aggregation and send data to closer nodes, which are their parents. Several intermediate levels may exist in a scale-free network.

Leaf nodes: These sensors are situated in the outmost levels of the network (from the base station). They collect data from the environment, process in-network aggregation and send data packets to their parents.

Figure 1. WSN organized as a scale-free network.

3 Quering Data in WSN Applications

3.1 Running Example

In order to illustrate the use of the proposed data model and of the query language SNQL, let us consider the following running example. An application consisting of an environmental biocomplexity mapping has the goal of relating data about temperature, pressure and humidity coming from sensors that are spatially distributed in an environment. Data are related according to delimited geographical areas. We also consider that all sensors in the network have similar capacity of processing, memory, battery power; and that all nodes are already organized in a scale-free network.

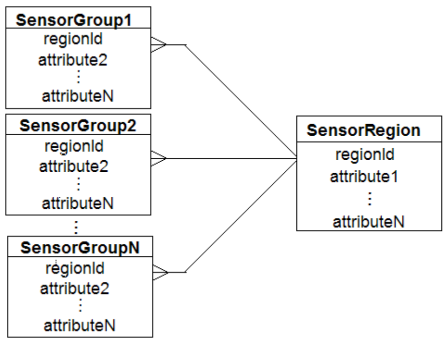
3.2 Data Model

In this section, we describe a generic data model for WSN applications. The applicability of the data model is illustrated by applying it to the running example (Section 3.1). The goal of using the proposed data model is to allow a logical view over data-streams dealt by the system. Therefore applications can see data flowing through a WSN as tuples of (virtual) relations. The data model abstracts the user from physical details such as identifying relevant sensors for a given query (i.e., querying data from an specific geographical region) [16], identifying data which have to be processed in sensor nodes or in the base station, and applying optimization rules to reduce the volume of transmitted packets for minimizing the power consumption and the traffic in the network. Furthermore, users can define declarative queries based on a data model, in the same way it happens in conventional database applications.

Each sensor group is represented by a specific relation (*SensorGroup1*, *SensorGroup2*, …, *SensorGroupN*) and has attributes related to the monitored phenomenon. This data model captures the information about geographical areas by means of the *SensorRegion* relation (see Figure 2), which contains an attribute (*regionId*) as an identifier for the delimited geographical area where a set of sensors collects data. This attribute is common to all relations, thus it allows information generated by different sensor groups be related through geographical areas (regions), where each region may have sensors belonging to different groups (see Figure 2).

Figure 3 shows the data model designed to the running example (Section 3.1). It is worth noticing that Temperature, Humidity and Pressure are in fact virtual tables seen over data stream flowing through the WSN. Observe that each relation has information related to the physical phenomenon represented, e.g. Temperature has information about the geographical area where data were collected (regionId), the measured temperature value (collectedValue), the number of collections performed (sampleAmount) and the considered scale (scale, which identifies if the measurement is made in Celsius or Fahrenheit degrees). In the same way, the SensorRegion relation has attributes to represent the properties of each region (see Figure 3).

A practical example where the proposed data model would be applied is to obtain the so-called Heat Index or THIndex. The Heat Index is a relation between temperature and humidity which has the purpose of studying health risks caused by such combination, since the higher the value for THIndex, the higher the risk of heat stroke. For example, if the temperature sensors register a temperature higher than 20ºC and, at the same time, a high value of humidity, the Heat Index will be higher than the air temperature itself – in this case, higher than 20ºC.



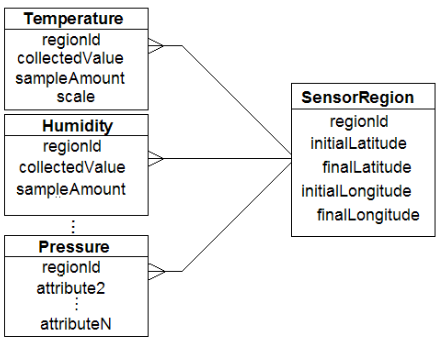


Figure 2. Data model for WSN applications. **Figure 3. Running example Data model.**

3.3 SNQL (Sensor Network Query Language)

In this section, we describe SNQL (Sensor Network Query Language), an SQL-like language specially designed for WSNs. This language includes five features for expressing and processing queries in WSNs. First, users can express declarative (ad-hoc) queries. Second, users can control the data volume and the precision of the results by previously defining parameterized features. Third, it supports continuous queries [2], that is, the results are continuously collected, updated, and sent back to the users. Fourth, users are able to deal with the notion of a predefined query, which means that the same query is continuously re-injected into the system in predefined periods. Finally, SNQL defines statements to adjust clause values of submitted queries, previously injected into the system, on-the-fly.

Table 1. Brief specification of SNQL clauses.

Clauses	Specification
SELECT {<expr>}	<expr>: It specifies a subset of the attributes in the original relations.
FROM {<sensor group>}	<sensor group>: It specifies the considered sensor group.
[WHERE {<pred>}]	<pred>: It specifies a set of predicates which filters processed tuples.
[GROUP BY {<exprgroup>} [HAVING {<predhaving>}]]	<exprgroup>: It specifies a subset of attributes in which aggregate functions are based. <predhaving>: It specifies predicates, based on aggregate functions, to filter aggregated results.
TIME WINDOW <twseconds> | CONTINUOUS	<twseconds>: It defines the time interval in which a query is valid to the system (e.g., 3600 s). CONTINUOUS means that the query validity time is infinite.
[DATA WINDOW<dwnumrows>]	<dwnumrows>: It specifies an exact amount of data to be collected.
SEND INTERVAL <sndseconds>	<sndseconds>: It specifies the interval between two consecutive sending of packets (e.g., 120 s).
SENSE INTERVAL <snsseconds>	<snsseconds>: It specifies the interval between two consecutive data collection (e.g., 10 s).
[SCHEDULE <numexecutions> [{datetime}] | CONTINUOUS]	<numexecutions>: It defines the number of times a query is injected into the system. If this number is greater than 1, a set of date and time values must be informed to determine when executions should occur (e.g., 2 '10-oct-05 14:00:00', '15-oct-2005 14:00:00'). CONTINUOUS means that the query has to be injected into the system an infinite number of times. Each time a different query instance is considered.

Sensors and the base station use the *Time window* and *Data window* clauses in order to know when they should stop a query execution. The predefined value *Continuous* in the *Time window* clause specifies a continuous query. In this case the query is executed for an indefinite period of time and results are continuously updated.

The *Sense interval* clause specifies the interval between consecutive data collections. High values for that interval means that less data will be collected and consequently results will be less precise. On the other hand, low values for *Sense interval* may produce more precise results, since a larger sample will be collected. However, such a scenario means more data processing and more data to be stored in memory. The *Send interval* clause value should also be carefully defined, since it impacts on memory usage. If its value is high, more data should be stored in sensor nodes, which increases the chance of memory overflows. But low values may produce larger amounts of small packets, increasing data traffic and also making sensors waste more energy. Thus observe that values for the *Sense interval* clause and *Send interval* clause directly influences network performance and sensor lifetime.

The *Schedule* clause works by allowing users to specify a number of times and the periodicity in which a query should be injected into the system. For instance, 2 '10-oct-05 14:00:00', '15-oct-2005 14:00:00', means that the query will be executed two times in the specified date and time, which results in two query instances. The predefined value *Continuous* forces the query to be executed an indefinite number of times. In this case, at the end of each query instance execution, all materialized data in the base station is discarded and new data materialization starts.

Discover what is the maximum temperature value and the correspondent minimum humidity value in the following delimited geographical region: 3º43'08"S-38º31'51"W and 3º43'16"S-38º31'14"W. Focus temperature values greater than 20^{o}C and humidity values smaller than 0.7. The query should be injected into the system 2 times (in 10-oct-05 at 14:00:00 and in 15-oct-05 at 14:00:00), each time it might stay executing for 3600 seconds. Data should be collected each 10 seconds and sent to the base station each 120 seconds.

```
SELECT r.regionID AS GeographicalRegion,
           MAX(t.collectedValue),  MIN(h.collectedValue)
FROM    Humidity h, Temperature t, SensorRegion r
WHERE r. regionID = t. regionID AND r. regionID = h. regionID
AND  r.initialLatitude    > 034308  AND r.finalLatitude < 034316
AND  r.initialLongitude > 383151  AND r.finalLatitude < 383114
AND  t.collectedValue  > 20
AND  h.collectedValue  < 0.7
GROUP BY             r.position
TIME WINDOW     3600
SEND INTERVAL    120
SENSE INTERVAL  10
SCHEDULE 2 '01-jan-06 14:00:00', '15-jan-06  14:00:00'
```

Figure 4. Query example written in SNQL.

SNQL also defines statements to adjust (on-the-fly) the following clause values: *Time window*, *Data window, Sense interval, Send interval* and *Schedule*. Thus, SNQL makes possible to inject a query fragment into the system (WSN) that informs a new value for any of those clauses. It is important to note that values for the *Sense interval* and *Send interval* clauses can be dynamically adjusted by ADAGA (see next section).

4 ADAGA – ADaptive AGgregation Algorithm

ADAGA was designed for processing in-network aggregation in sensor nodes. Furthermore, this algorithm explores techniques in order to adapt its behavior according to memory and energy usage (very limited resources in sensor nodes). The goal is to achieve better approximate results in resource constraint situations. ADAGA presents the following features:

In-network aggregation: this feature reduces the amount of data to be transmitted by sensor nodes, which consequently reduces power consumption in sensors and data traffic in WSNs. The idea is to aggregate data as they are flowing through the network in such a way that packets of the same sensor group have their data aggregated in order to produce a unique and compacted data packet. Furthermore, by using in-network aggregation less data is locally stored thus reducing memory usage.

Monitoring energy consumption and memory usage: ADAGA monitors energy consumption and memory usage in order to adjust its behavior by means of reducing the activities performed by sensors. The key idea is to dynamically adapt values for *Sense interval* and *Send interval* clauses, according to available energy and memory (See Section 4.2). For instance, when values for *Sense interval* and *Send interval* are increased, less sensing activity, data processing and data storage is performed. Thus, sensors lifetime (regarding available battery) and available memory space are increased. There is extra processing due to resource monitoring. However, the benefit of extending sensor node lifetime by using a resource-aware strategy is more significant than the extra processing drawback.

Better result approximation: The ADAGA deals with resource constraints (energy and memory) by dynamically adjusting the *Sense interval* and the *Send interval* clause values. However, when those values are reduced, the amount of collected data is reduced as well. For that reason, a challenge when running ADAGA is to produce the best approximate results possible (even having smaller samples to produce results), when resource contraints are faced. Section 4.2 explains the strategy implemented by ADAGA to achieve that goal and Section 4.4 evaluates that strategy.

Fault Tolerance: Since sensors are prone to failures, it is important that packets have more than one way to reach the base station. ADAGA supports packet replication and also avoids the data duplication in result that may be caused by packet replication (see Section 4.3).

4.1 Algorithm

ADAGA is executed in five sequential stages. Sensors, such as Mica Motes [13], are not able to perform parallel operations; others, such as μAMPS, can receive and send data simultaneously, but other operations also cannot be executed in parallel [5]. The proposed algorithm uses three logical data structures (lists) to temporarily store data: a receiving area, which stores received packets; a processing area, where data to be aggregated is stored; and a sending area, where packets to be sent to other nodes are stored. ADAGA also admits packet replication by adopting a routing strategy which progressively eliminates replicated packets, as they are passed though the network, in order to produce better approximate results. The routing strategy is presented in Section 4.3. The five stages of ADAGA are as follows (see Figure 5):

Stage 1: The key goal of the first stage is to control the other four stages of the execution. Basically, it has a sequence of nested loops, where the first one (line 1) is performed *i* times, where *i* is the number of query executions (defined in the *Schedule* clause); the second loop (line 3) specifies that each query has to be executed while the *Time window* clause value is not reached; the third loop (line 6) specifies that the *Send interval* clause value should not overtake the *Time window* clause value.

Stage 2: This stage is responsible for processing data temporarily stored in the processing area. In other words, it performs in-networking aggregation which consists of filtering data, according to query predicates, and aggregating similar data (packets generated by nodes from the same sensor group).

Stage 3: It is responsible for monitoring energy and memory usage. This stage adjusts data collection according to resource availability (energy and memory) in order to produce results consistent with real results (when no resource constraints is faced). Section 4.2 describes the procedures *adaptSendInterval(x1,a,t)* (line 6) and *adaptSenseInterval(x2,b,a´)* (line 7) in more details.

Stage 4: This stage deals with received packets stored in the receiving area (line 3). If a packet and a sensor node, which received it, have the same sensor group (line 4), the packet is stored in the processing area in order to be aggregated with locally collected data (line 7), otherwise, the packet is stored in the sending area (line 10).

Stage 5: It is responsible for sending packets to parent nodes. For each packet *p* stored in the sending area (line 1), this stage verifies the number of copies of *p* (*c* in line 3), in the packet header; and the number of local copies of *p* (*l* in line 4). The result value for n = *c* – *l* is obtained in line 6. Thus if n > 1 (line 8), the packet is sent to a randomly chosen parent (line 10), just as it happens in the *Gossiping* approach [8]. Otherwise, it is sent to all parents (line 12). When each send interval is reached, data sensing is interrupted, the data stored in memory are packed and sent to parent nodes together with packets received from other sensor nodes. Data sensing resumes at the end of Stage 5.

```
Stage 1: Adaga (Sensor s, Query q)
1:  For i = 1 to number of query executions do
2:    initialize timerTimeWindow;
3:    While timerTimeWindow < q.TimeWindow do
4:      initialize timerSendInterval;
5:      s.SenseInterval ← q.SenseInterval
6:      While timerSendInterval < q. SendInterval do
7:        initialize timerSenseInterval;
8:        If timerSenseInterval < q. SenseInterval then
9:          d ← collectDataFromTheEnvironment();
10:         processingArea ← d;
11:       End if;
12:       processData();
13:     End while
14:     receivePackets (Sensor s);
15:     sendPackets (Sensor s);
16: End for

Stage 2: ProcessData()
1: p1 ← get a packet from the processing area;
2: p2 ← get a packet from the processing area;
3: filter data in p1 and p2, according whith predicates in the query;
4: p3 ← agg(p1,p2); //apply aggregate function specified in the query
5: //and generate p3 which substitutes p1 and p2;

Stage 3: MonitorResource(Sensor s, Query q)
1: x1 ← s.getAvailableEnergy();
2: x2 ← s.getAvailableMemory();
3: a ← q.SendInterval;
4: t ← q.TimeWindow;
5: b ← q.SenseInterval;
6: a´ ← s.SendInterval ← adaptSendIntervals ( x1, a , t );
7: s.SenseInterval ← adaptSenseInterval ( x2, b, a´ );

Stage 4: ReceivePackets (Sensor s, Query q)
1: For each packet in the receivingArea do
2:   MonitoResource(s,q);
3:   p ← get a packet from the receivingArea;
4:   If p.sensorGroup = s.sensorGroup then
5:     //p.amountOfCopies=nº. of copies in the p header
6:     If p.amountOfCopies = 1 then
7:       processingArea ← p
8:     End if
9:   Else
10:    sendingArea ← p;
11:  End if
12: End for

Stage 5: SendPackets(Sensor s)
1:  For each packet in the sendingArea do
2:    p ← get a packet from the sendingArea;
3:    c ← p.amountOfCopies; // registered in the p header
4:    l ← get the number of local copies of p;
5:    discard l - 1 copies of p;
6:    n ← c - l;
7:    register n in p header as the new number of p copies;
8:    If n > 1 then
9:      s´ ← parent of s, chosen randomicaly;
10:     send p to s´;
11:   Else
12:     send p to all parents of s;
13:   End if
14:   discard p from the sendingArea;
15: End for
```

Figure 5. ADAGA algorithm.

SNQL supports the following aggregate functions: *Count*, *Sum*, *Max*, *Min*, *Average* and *Median*. Gray *et al.* [7] classify those aggregate functions in the following categories: (*i*) *Distributive* (*Max*, *Min*, *Count* and *Sum*), (*ii*) *Algebraic* (*Average*) and *Holistic* (*Median*). In *Distributive* aggregates, the size of each partial result, obtained by applying an aggregate function in sensor nodes, is the same as the size of the final result calculated in the base station. For instance, when calculating the function *Max*, the size of any partial result is 1, which corresponds to the same size of the final result. For the *Algebraic* aggregate *Average*, the partial results obtained in each sensor node corresponds to the distributive functions *Sum* and *Count*, applied to the collected data. In this case, the final result for *Average* is produced in the base station. In *Holistic* aggregates, all the data must be brought together to be aggregated by the base station, since no useful partial aggregation can be produced in sensor nodes. ADAGA implements all the aggregate functions belonging to the three aforementioned categories by applying the function *agg(p1,p2)*, specified in line 4 of Stage 2 (Figure 5).

In order to show how ADAGA aggregates data w.r.t different aggregate functions let us look at the following scenario. Consider two sensor nodes, say *A* and *B*, belonging to the sensor group *Temperature*. Whenever nodes *A* or *B* receive a packet produced (and sent) by a node belonging to the sensor group *Temperature*, they open the packet and aggregate data transported by the packet with locally collected data, according to the aggregate function (specified in the query). Thereafter, they send the packets to the base station. Suppose that *A* and *B* produce packets p_A and p_B, respectively. Table 2 illustrates aggregated data by different aggregate functions in ADAGA. Observe that the size of a packet depends on the aggregate function specified in the query. For example, after the sensor node *A* has collected the values 20° - 20° - 21° (*A* might have received them from other temperature sensors or might have sensed them locally), the packet p_A contains just the value for *Sum* (i.e., 61) and the value for *Count* (i.e., 3), if *Average* were the aggregate function specified in the query injected into the WSN. On the other hand, if *Median* were the specified function, the packet p_A should contain the set {20°, 20°, 21°} of temperature values.

Table 2. Partial results obtained by applying different aggregate functions

Node	Collected data	Aggregate function					
		Min	*Max*	*Count*	*Sum*	*Average*	*Median*
A	20° - 20° – 21°	20°	21 °	3	61	61;3	20° - 20° - 21°
B	21° - 21° – 22°	21°	22 °	3	64	64;3	21° - 21° - 22°
Base Station	p_A and p_B*	20°	22 °	6	125	125/6 = 20.83	21°

*p_A is the packet produced by node *A* and p_B is the packet produced by node *B*

4.2 Monitoring Sensor Resources

ADAGA acts in a proactive way, since it monitors energy and memory usage in order to dynamically adjust sensor activities to these resources availability (energy and memory). To achieve that goal, ADAGA works with two strategies: (*i*) adjusting the *Send interval* clause, in case of energy constraints; and (*ii*) adjusting the *Sense interval* clause, in case of memory constraints. These two strategies are implemented by applying the functions *f(v)* and *h(m)*, which correspond to the *adaptSendInterval(x1,a,t)* and the *adaptSenseInterval(x2,b,a´)* procedures (see Section 4.1), defined as follows.

The first strategy uses a function *f(v)* which adjusts the *Send Interval* clause value (*d*) according to energy (battery) availability (*v*). Since almost 50% of power consumption in a sensor node occurs due to communication activities (sending and receiving data) [13], battery is a critical resource to sensors. For that reason, ADAGA may delay the sending of packets in order to save energy and increase sensor lifetime. In other words, ADAGA increments the value of *Send Interval* to reduce the frequency of packets transmission.

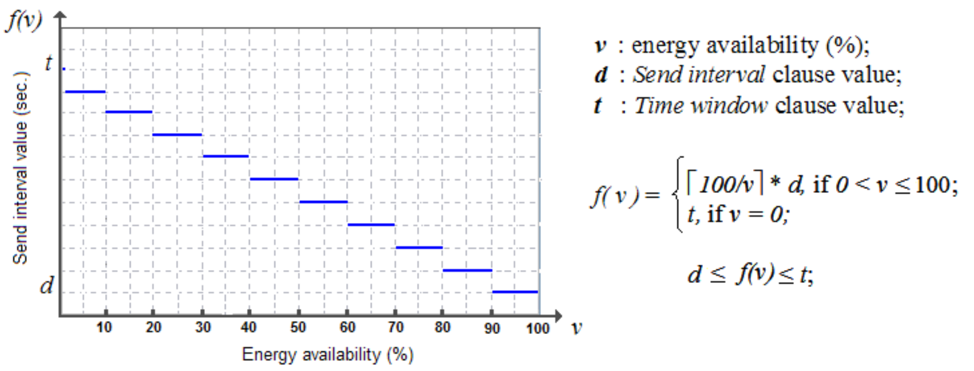


Figure 6. Function for adapting the send interval clause to energy usage.

Figure 6 shows how the value for the *Send interval* clause is incremented according with energy availability, *v*. The function *f(v)* varies from the minimum, which correspond to the value, *d*, defined in the *Send interval* clause, to the maximum, which is the value for the *Time Window* clause, *t*. Note that if the available energy is close to 100%, *f(v)* returns the value *d*. However, as the energy is consumed, *f(v)* is progressively increased. Thus, packet sending is postponed of $\lceil 100/v \rceil$ units of time, where $\lceil 100/v \rceil \in \mathbb{N}^*$ and $1 \leq \lceil 100/v \rceil \leq t/d$. Finally, if energy availability is close to 0%, *f(v)* assumes the value *t*, which means that the sensor node is not able to work, because no energy is available. Observe that *f(v)* should not be continuous, since sensor nodes are supposed to send and receive packets in predefined periods of time (periods known by all sensor nodes in the network).

The second strategy consists of adjusting the *Sense interval* clause value, *g*, according to memory availability, *m*. The goal is to dynamically decrease sensing activity as memory availability decreases, which reduces in turn the sample size (number of collections made from the environment). Accordingly, less data should be stored in memory to be aggregated and to be sent to the base station. Furthermore, power consumption associated to sensing and data processing activities is also decreased. ADAGA adjusts the value for the sense interval according with the function *h(m)*. The function *h(m)* varies from the minimum, which correspond to value of the *Sense interval* clause, *g*, to the maximum, the value of the *Send interval* clause (obtained by applying the *f(v)* function); both clauses defined in the query.

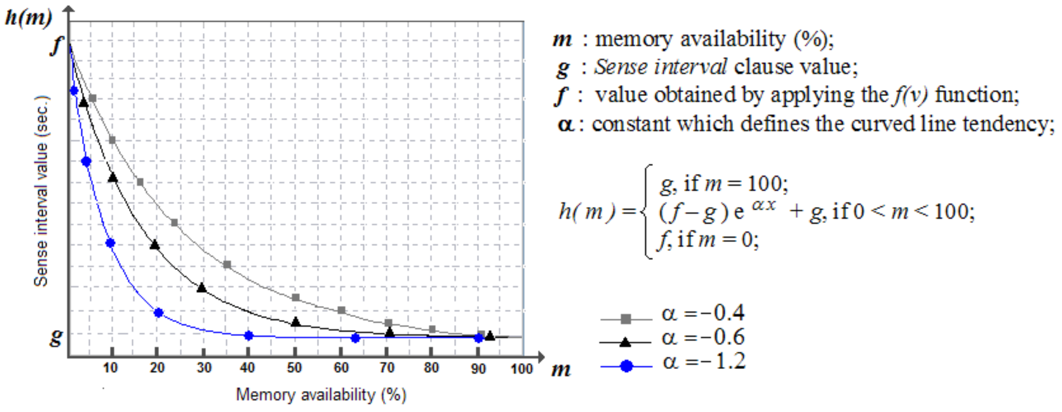


Figure 7. Function for adapting the sense interval to memory usage.

Figure 7 shows three possible curves for the function *h(m)*, each of which having a different value for α, which defines the curved line tendency. For α=-0.4, *Sense interval* adjustments occur too soon (when 90% of memory is available), which may change more quickly the value defined by the user for the *Sense Interval* clause. On the other hand, for α=-1.2, *Sense interval* adjustments occur too late (just when one has about 30% of available memory), which may represent a significant amount of memory usage. We consider an intermediate value, α=-0.6, in our experiments. In this case the function *h(m)* stays relatively constant between 50% and 100% of memory availability and thus the sensor node has *h(m)* equal or close to *g*. However, as memory usage increases (memory available decreases), the value for *h(m)* also increases. If the available memory reaches 0%, the sense interval value assumes the value obtained in *f(v)*, which means that no collection will be made until the next send interval is reached. It is important to note that after each sending of packets, *h(m)* assumes the value *g* again.

In general, approaches for in-network aggregation in WSN applications work in a reactive fashion. Those approaches interrupt the sensing activity when memory overflows or the battery power exhausts. The goal of using the functions *f(v)* and *h(m)* is to make sensor nodes self-configurable devices by proactively monitoring the resource usage and adjusting sensor node activities. Indeed, the *Sense interval* and *Send interval* value adjustments made by ADAGA has priority over the values defined by users (in a SNQL query).

4.3 In-network aggregation with packet replication

In a WSN, leaf nodes collect and pack data to be sent to other nodes. Intermediate nodes collect data (by sensing) and receive packets from other sensors. Since there would probably be more packaged data in intermediate nodes, failures in these nodes are more critical. In order to reduce the impact of packet losses in case of node failure, alternative packet routes should exist, which means that a sensor should send a packet to several sensors close to it. In other words, replication is necessary. Nevertheless, replication should be limited in order to avoid the implosion deficiency faced by *flooding* protocols [9]. Implosion consists of sending the same packet to several nodes, which also send this packet to several other nodes. Packet replication may become an even more difficult problem to solve when in-network aggregation is considered, since data packets should be aggregated progressively in each node they passed through.

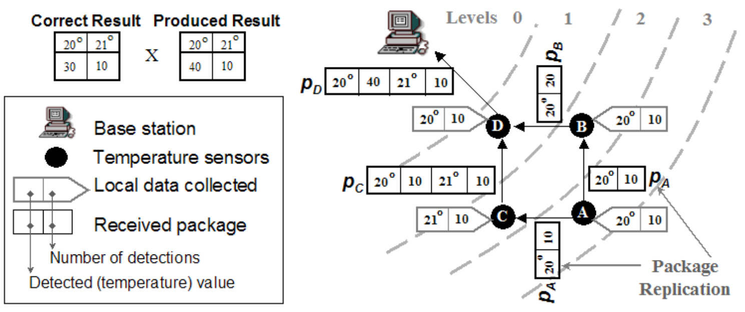


Figure 8. Data replication happens when a node has more that one parent.

In Figure 8, we have a scenario where a set of temperature sensors, represented by nodes *A*, *B*, *C* and *D* are distributed across three levels of a scale-free network. Suppose that each sensor node performs 10 collections (detections) from the environment before sending packets to its parents. Each packet is composed by an ID and a set of tuples of the form: detected (temperature) value and the number of detections. Now, suppose that the node *A* replicates a packet p_A by sending it to both *B* and *C*. In turn, *B* aggregates its local collected data (i.e., 20° – 10, which means the value 20 has been detected 10 times) with data enclosed in p_A (which has been sent by *A*), generating a new packet p_B. *C* aggregates its local collected data (21° - 10) with data come from *A*, generating a new packet p_C. *D* aggregates its local collected data (20° - 10) with data coming from *B* and *C*, but it is not able to detect data replication because the initial identification of the packet generated in *A* was lost. Finally, the base station produces the final result, which does not reflect the correct result (see Figure 8).

In order to avoid such a problem, ADAGA admits packet replication but not as it happens in *flooding* strategy. For that reason, we have developed an alternative packet-replication strategy in order to run ADAGA. When a packet *P* finds the first node (*N*) that has more than one parent, a copy of *P* is generated to each parent of *N*. After that, when one of the copies of *P* finds a node *N'*, which also has more that one parent, just one of those parents is randomly chosen to receive *P*. Therefore, after the first packet replication, no other copy for *P* is generated any more. In order to make that control, when a packet is generated, the number of replicas (copies) generated, *c,* is stored in its header. There are two possibilities when a node *N*, which has more than one parent, receives a packet *P*: *(i)* if *c* = 1, *P* is replicated for each of the *N's* parent; *(ii)* on the other hand, if *c* > 1, *P* is sent to just one of the *N's* parents (which is randomly chosen). Therefore, after the first packet replication, the number of copies for a given packet *P* is not increased any more.

When a node *N* receives *m* copies of the same packet *P*, the copy of *P* (P_k) which has the smallest value for *c* is kept, the other *m* - 1 copies are discarded, and *c* is updated, i.e., *c* is set to *c* = *c* - (*m* - 1). The new value for *c* is then stored in the header of P_k. Observe that, when the value *c* = 1, the data in the packet (detected value and the number of detections) can be aggregated because there are no risk of generating duplicate in final result. Hence, packet copies are progressively discarded as they are passed through the network and data in given packet is aggregated when there are no replicas for the packet (*c*=1).

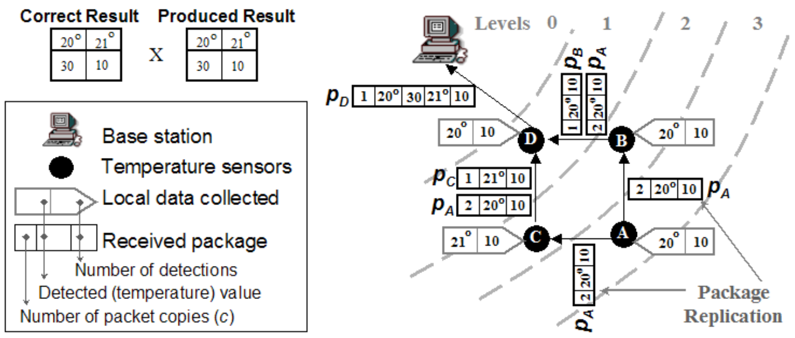


Figure 9. In-network aggregation with the ADAGA algorithm.

In order to illustrate how ADAGA processes in-network aggregation, consider the same example depicted in Figure 8. Observe in Figure 9 how data are progressively aggregated until they reach the base station (delays are disregarded). In Figure 9, the header of a packet contains the number of packet copies (*c*). Suppose that node *A* collects data and generates the packet p_A which is sent to its parent nodes *B* and *C*. Since the packet p_A has 2 copies (i.e., *c* > 1), its data are not aggregated with data collected in nodes *B* and *C*. Since the node *D* receives 2 copies of p_A, *D* discards one of them and set *c*=1, according to the routing strategy described above. Thereafter, *D* aggregates data in the packet p_A with the locally collected data. Finally, the result packet p_D is generated by *D* and sent to the base station.

Clearly, some packets received by the base station may still have copies left in the network. Similar to what happens in sensors, packet content cannot be processed until all their replicas are discarded. However, if a packet *P* in the base station has lost copies, its content would never be aggregated because the *c* would never be 1. There are two alternatives to overcome that problem. The first consists of using timeouts to indicate when a packet can be opened by the base station, even if *c* > 1. The second is to wait until the query validity time (*Time window* clause) is reached, since data will not be received any more, the packet can be opened because its copies will not be accepted if they arrive later. However if the query is defined as a continuous query, the second alternative cannot be applied because the query validity time is infinite – packets with lost copies in the network would never have their data aggregated in the base station.

4.4 Evaluation

We evaluated the ADAGA by executing *Average*, *Max*, *Min*, *Sum*, *Count* and *Median* operations in a sensor network simulator, developed in C++, and executed in a Pentium IV machine. This simulator allows for the configuration of memory and energy availability. Thus, it is possible to simulate how the ADAGA algorithm would response in resource constraint situations. In Figure 10 and Figure 11 we consider a SNQL query *Q* with the following clause values: *Time window* = 60 sec., *Send interval* = 60 sec., and *Sense interval* = 100 msec. *Q* is executed in order to collect temperature measurements from the environment. We also consider a uniform distribution of temperature values between –10^{o}C and 40^{o}C. Figure 10 shows how results (for the *Average* operation) assume different approximations of the correct result (when no energy or memory constraint is faced), based on three approaches: (*i*) *proactive*, explored in ADAGA by monitoring memory and energy availability, in order to adjust sensing activity; *(ii) reactive*, which consists of processing in-network aggregation without monitoring resource usage; and *(iii)* a *conventional* for WSN applications, which consists of just collect and send data to other nodes, *without in-network aggregation*. Figure 11 shows the amount of data produced in a sensor node by different aggregate operations with ADAGA and without applying in-network aggregation.

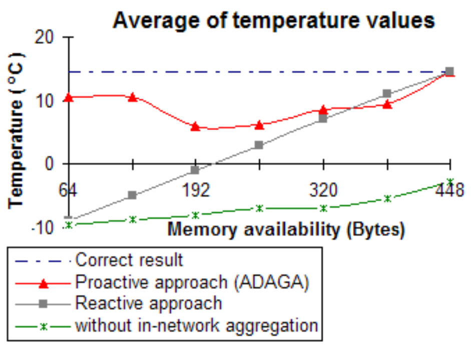


Figure 10. Calculation of operation *average* applied by a sensor.

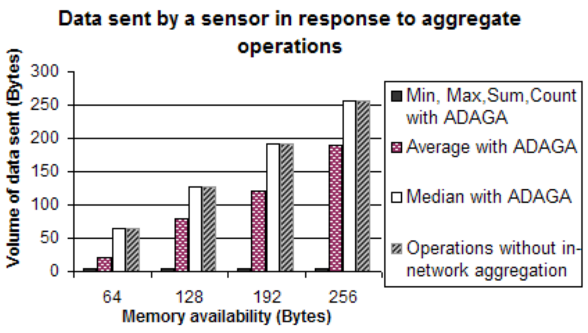


Figure 11. Volume of data sent by a sensor according with memory availability.

Note that in 60 seconds, 600 collections would have been made, if memory constraints were disregarded producing the result for the average operation of 14.53^{o}C. By using in-network aggregation, a maximum of 408 bytes would be necessary to store distinct detected values, and the number of detections of each value. In Figure 10, we observe that ADAGA produces better estimations of the correct results, considering different levels of memory constraints, than approximations produced by a reactive approach or when in-network aggregation is not applied. In fact, the proactive approach curve (see Figure 10) experiences decreases when it does not perform the collection of a temperature value which would strongly influence the average operation result. As an extension, we are planning to estimate non-collected data (because of *Sense interval* clause adjustments), based on statistical methods.

Figure 11 shows the estimated amount of data that would be sent to the base station by a certain sensor node. Note that just one integer value is necessary to store *Min*, *Max*, *Sum*, and *Count* operation result. On the other hand, results produced by an *Average* operation are dependent of the number of distinct collected values. The strategies that cannot use in-network aggregation (i.e. Median) force sensors to send all data collected to the base station, and thus sensors are more prone to resource constraints.

5 Related Work

There is a growing interest in query processing for WSNs. Some researches have explored in-network aggregation as an alternative to achieve energy efficiency when propagating data from sensors to sink nodes [11][12][13][16]. In-network aggregation approaches are mainly differentiated by their network protocols for routing data. Directed diffusion is proposed in [10] as a data-centric communication where a sink node broadcasts an interest that describes the desired data to its neighbors. As interests are passed throughout the network, gradients are formed indicating the direction in which collected data will flow back. Each node maintains a small cache of recently received data items in order to avoid duplicates. However, maintaning an extra cache represents aditional overhead. In [11] a greedy incremental tree (GIT) is proposed, in which a shortest path is established for only the first source to the sink node and the others are incrementally connected at the closest point on the existing tree, forming a rigid architecture. TAG [12] works with a routing tree rooted at a base station and does not accept duplicate packets in the network. When a node has two or more parents, aggregated values are divided by the number of parents and sent to them. A drawback of this approach is that if a sensor fails, its data will be lost. ADAGA uses a routing protocol for a scale-free network that performs in-network aggregation. In Section 4.3, we have shown that even if duplicate packets are admitted in some nodes, data are not duplicated in the result because the copies are progressively eliminated as they are passed through the network and data are only aggregated when there are no copies. Furthermore, this algorithm is also resource-aware, since it adapts its behavior to energy and memory constraints.

Some works have proposed extensions to the conventional SQL as an approach to work with declarative queries in WSN applications. The TinyDB Project at Berkeley proposed the acquisitional query language [13], which has some simple extensions to SQL for controlling data acquisition. Madden et al show how acquisitional issues influence query optimization, dissemination, and execution [13]. The proposed query language for TinyDB has clauses with similar semantic to the SNQL clauses *Time window, Sense interval* and *Schedule*. Nonetheless, we claim that SNQL gives more flexibility to applications, since it allows changing clause values on-the-fly. For example, besides supporting continuous queries, SNQL allows that intervals between two consecutive sending of packets can be modified during the query execution (by modifying the value of *Send interval* clause).

6 Conclusion

In this paper we described some important issues related to sensor networks and in-network aggregation. We propose an approach for query processing in wireless sensor networks consisting of a query language, called SNQL, and an adaptive algorithm, ADAGA. SNQL has some clauses especially designed to support application needs in wireless sensor networks. We argued that it is a flexible query language because it supports declarative queries, which can have some of its clauses changed on-the-fly. Furthermore, it supports continuous queries. We consider a scale-free network scenario, in which a base station receives data from groups of sensors. We propose the ADAGA algorithm for query processing in sensor devices. It performs in-network aggregation and adapts its behavior under energy and memory constraints. We believe that these contributions form a consistent approach for query processing in wireless sensor networks, which is a fruitful research area for the database community.

References

[1] Akyildiz, I., Su, W., Sankarasubramaniam, Y. And Cyirci, E. *Wireless sensor networks: A survey*. Computer Networks, vol. 38, no. 4, pp. 393-422, March 2002.

[2] Babu, S. and Widom, J. *Continuous Queries over Data Streams*. SIGMOD Record, Vol. 30, No. 3, September 2001.

[3] Barabasi, A. and Albert, R. *Emergence of scaling in random networks.* Science, 8, October 1999.

[4] Considine, J., Li, F., Kollios, G. and Byers, J. *Approximate Aggregation Techniques for Sensor Databases*. In Proceedings of ICDE, April 2004.

[5] Cho, S., Shih, E., Ickes, N., Min, R. et al. *Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks*, ACM/IEEE MOBICOM, 272–287, Italy, July 2001.

[6] Golab, L., Özsu, M. T. *Issues in Data Stream Management*. ACM SIGMOD, volume 32, No. 2. University of Waterloo, Canada, June 2003.

[7] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. *Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total*, February 1996.

[8] Hedetnieme, S., Hedetnieme, S., and Liestman, A. *A Survey of Gossiping and Broadcasting in Communication Networks*. Networks vol .18 pp.319-349, 1988.

[9] Heinzelman, W., Kulik , J. and Balakrishnan, H. *Adaptive Protocols for Information Dissemination in Wireless Sensor Networks*. ACM/IEEE MOBICOM, August 1999.

[10] Intanagonwiwat, C., Govindan, R. and Estrin, D. *Directed diffusion: A scalable and robust communication paradigm for sensor networks*. ACM MOBICOM ACM, August 2000.

[11] Intanagonwiwat, C., Estrn, D., Govindan, R. and Heidemann, J. *Impact of Network Density on Data Aggregation in Wireless Sensor Networks*. International Conference on Distributed Computing Systems (ICDCS),p.457, July 2002.

[12] Madden, Samuel R., Franklin, Michael J., Hellerstein, Joseph M. and Hong, W. *TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks*. OSDI, December 2002.

[13] Madden, Samuel R., Franklin, Michael J. and Hellerstein, Joseph M. *TinyDB: An Acquisitional Query Processing System for Sensor Networks*. ACM Transactions on Database Systems, Vol. 30, No. 1, Pages 122-173. March 2005.

[14] Solis, I., Obraczka, K. *In-Network Aggregation Trade-offs for Data Collection in Wireless in Sensor Networks*. INRG Technical Report 102, August 2003.

[15] Tubaishat, M., Yin, J., Panja B. and Madria, S. *A Secure Hierarchical Model for Sensor Network.* ACM SIGMOD, volume 33, No. 1, March 2004.

[16] Yao, Y. and Gehrke, J. *Query Processing for Sensor Networks*. In proceedings of the CIDR Conference. 2003.