An adaptive in-network aggregation operator for query processing in wireless sensor networks

Angelo Brayner a,*, Aretusa Lopes a, Diorgens Meira a, Ricardo Vasconcelos a, Ronaldo Menezes b

a Department of Computer Science, University of Fortaleza (UNIFOR), Brazil
b Department of Computer Science, Florida Institute of Technology, USA

Available online 5 July 2007

Abstract

A wireless sensor network (WSN) is composed of tens or hundreds of spatially distributed autonomous nodes, called sensors. Sensors are devices used to collect data from the environment related to the detection or measurement of physical phenomena. In fact, a WSN consists 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). Sensors are limited in power, computational capacity, and memory. Therefore, a query engine and query operators for processing queries in 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 query operators (algorithms) can adjust their behavior in response to specific events that take place during data processing. In this paper, we propose an adaptive in-network aggregation operator for query processing 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. ADAGA can correctly aggregate data in WSNs with packet replication. Moreover, ADAGA is able to predict non-performed detection values by analyzing collected values. Thus, ADAGA is able to produce results as close as possible to real results (obtained when no resource constraint is faced). The results obtained through experiments prove the efficiency of ADAGA.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Wireless sensor networks; Adaptive in-network aggregation; Query processing in WSNs

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 (Akyildiz et al., 2002).

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 (Tubaishat et al., 2004). 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 (Barabasi and Albert, 1999), 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 (Golab et al., 2003). 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 through the network — this is called in-network aggregation (Considine et al., 2004; Madden et al., 2002; Solis et al., 2003). This technique reduces the amount of data to be transmitted by sensors and consequently the data traffic in the network.

Query-processing operators 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 operator, called ADAGA (ADaptive AGgregation Algorithm for sensor networks), for processing in-network aggregation in WSNs. ADAGA is an adaptive operator to dynamically adjust sensor activities (data sending and data collection), by monitoring energy consumption and memory usage, in order to maximize sensor node lifetime and also the accuracy of query results, mainly when sensor nodes experience energy or memory constraints. The goal is to make sensor nodes self-configurable devices by proactively monitoring the resource usage. ADAGA presents the following key features:

(i) Adaptive behavior. ADAGA adapts its behavior according to memory and energy usage by dynamically adjusting data-collection and data-sending time intervals;

(ii) In-network aggregation with packet replication. In order to reduce the impact of packet losses, a sensor should send a packet to several sensors close to it. In-network aggregation in WSNs with packet replication may produce an undesirable side-effect, since data packets should be aggregated progressively in each node they passed through. However, ADAGA can correctly aggregate data in WSNs with packet replication;

(iii) Prediction of non-sensed values. ADAGA is able to predict non-performed detection values by analyzing collected values. Accordingly, ADAGA is able to produce results as close as possible to real results (obtained when no resource constraint is faced).

We also briefly 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. ADAGA uses the clauses of SNQL as input parameters, since it adjusts the Sense interval and the Send interval clause values.

This paper is organized as follows. Section 2 briefly describes an abstract model of a wireless sensor network we are assuming in this paper. Section 3 describes our data model and gives an overview of the SNQL query language. In Section 4, the ADAGA operator is presented and evaluated. Section 5 discusses related work and Section 6 concludes the paper.

2. Wireless sensor networks

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 (Akyildiz et al., 2002): (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 (Akyildiz et al., 2002): (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 (Cho et al., 2001; Madden et al., 2005). 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 (Madden et al., 2005), 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 (Intanagonwiwat et al., 2002; Madden et al., 2002; Solis et al., 2003).

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 different sensing activities will be categorized in 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 Fig. 1.

3. Querying 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) (Yao and Gehrke, 2003), 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 Fig. 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 to be related through geographical areas (regions), where each region may have sensors belonging to different groups (see Fig. 2).

Fig. 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 sensed (detected) temperature value (collectedValue), the number of detections for CollectedValue (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 Fig. 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

![Fig. 1. WSN organized as a scale-free network.](image1)

![Fig. 2. Data model for WSN applications.](image2)
SNQL defines statements to adjust clause values of submitted queries, previously injected into the system, on-the-fly. 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 the 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, in the query depicted in Fig. 4, the parameter 2 ‘10-Oct-06 14:00:00’, ‘15-Oct-06 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.

SNQL also defines statements to adjust (on-the-fly) the following clause values: Time window, Data window, Sense 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 35 °C 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 35 °C.

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. SNQL includes five features for expressing and processing queries in WSNs, which can be observed in Table 1. 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 (Babu and Widom, 2001), 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, 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, in the query depicted in Fig. 4, the parameter 2 ‘10-Oct-06 14:00:00’, ‘15-Oct-06 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.

SNQL also defines statements to adjust (on-the-fly) the following clause values: Time window, Data window, Sense

---

Table 1

<table>
<thead>
<tr>
<th>Brief specification of SNQL clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clauses</strong></td>
</tr>
<tr>
<td>SELECT {&lt;expr&gt;}</td>
</tr>
<tr>
<td>FROM [&lt;sensor group&gt;]</td>
</tr>
<tr>
<td>[WHERE [&lt;pred&gt;]]</td>
</tr>
<tr>
<td>[GROUP BY {&lt;expgroup&gt;}</td>
</tr>
<tr>
<td>TIME WINDOW &lt;twseconds&gt;</td>
</tr>
<tr>
<td>[DATA WINDOW &lt;dwnumrows&gt;]</td>
</tr>
<tr>
<td>SEND INTERVAL &lt;sndseconds&gt; [FIXED</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SENSE INTERVAL &lt;snsseconds&gt; [FIXED</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[SCHEDULE &lt;numexecutions&gt;</td>
</tr>
</tbody>
</table>

---

Fig. 3. Running example Data model.
Discover what is the maximum temperature value and the correspondent minimum humidity value in the following delimited geographical region: 3°43'08"S-38°31'51"W and 3°43'16"S-38°31'14"W. Focus temperature values greater than 20°C and humidity values smaller than 0.7. The query should be injected into the system 2 times (on 10-Oct-06 at 14:00:00 and on 15-Oct-06 at 14:00:00), each time it might stay executing for 3,600 seconds. Data should be collected each 10 seconds and sent to the base station each 120 seconds.

Fig. 4. Query example written in SNQL.

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. The action of adapting the value of Sense interval is carried out according to available memory. On the other hand, the value of the Send interval clause is adapted according to energy availability (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 constraints are faced. Sections 4.2 and 4.3 explain the strategy implemented by ADAGA to achieve that goal and Section 4.5 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 4.4).

4.1. Algorithm

ADAGA is executed in five sequential stages. Sensors, such as Mica Motes (Madden et al., 2005), 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 (Cho et al., 2001). 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 through the network, in order to produce better approximate results. The routing strategy is presented in Section 4.4. The five stages of ADAGA are as follows (see Fig. 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

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 t.initialLatitude > 034308 AND t.initialLongitude < 383151
AND h.initialLongitude > 383114 AND h.initialLatitude < 383114
AND h.collectedValue > 20
AND t.collectedValue < 0.7
GROUP BY r.position
TIME WINDOW 3600
SEND INTERVAL 120
SENSE INTERVAL 10
SCHEDULE 2 '10-Oct-06 14:00:00', '15-Oct-06 14:00:00'

Fig. 5:
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 are faced). Section 4.2 describes the procedures adaptSendInterval($x_1, a, t$) (line 6) and adaptSensorInterval($x_2, b, a'$) (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 (Hedetniemi et al., 1988). 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.

SNQL supports the following aggregate functions: Count, Sum, Max, Min, Average and Median. Gray et al. (1996) 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 $\text{Max}$, 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. Additionally, Madden et al. (2002) also consider the class of count distinct aggregate function, which returns the number of distinct values collected by all sensor nodes. ADAGA implements all the aggregate functions belonging to the aforementioned categories by applying the function $\text{agg}(dt)$, specified in line 4 of Stage 2 (Fig. 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 $\text{Temperature}$. Whenever nodes $A$ or $B$ receive a packet produced (and sent) by a node belonging to the sensor group $\text{Temperature}$, 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\(^\circ\)–20\(^\circ\)–21\(^\circ\) (\( A \) might have received them from other temperature sensors or might have sensed them locally), the packet \( p_A \) contains just the value for \( \text{Sum} \) (i.e., 61) and the value for \( \text{Count} \) (i.e., 3), if \( \text{Average} \) were the aggregate function specified in the query injected into the WSN. On the other hand, if \( \text{Median} \) were the specified function, the packet \( p_A \) should contain the set \{20\(^\circ\), 20\(^\circ\), 21\(^\circ\)\} of temperature values.

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 \textit{Send interval} clause, in case of energy constraints; and (ii) adjusting the \textit{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 \textit{adaptSendInterval}(\( x_1 \), \( a \), \( t \)) and the \textit{adaptSenseInterval}(\( x_2 \), \( b \), \( a' \)) procedures (see Section 4.1), defined as follows.

The first strategy uses a function \( f(v) \) which adjusts the \textit{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) (Madden et al., 2005), 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 \textit{Send Interval} value to reduce the frequency of packets transmission.

Fig. 6 shows how the value for the \textit{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 \textit{Send interval} clause, to the maximum, which is the value for the \textit{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 \([100/v]\) units of time, where \([100/v]\) \( \in \mathbb{N}^\ast \) and \([100/v]/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). In this paper, delays in time synchronization are disregarded. The reader is referred to Zhao and Guinas (2004) for a deep discussion on time synchronization issues in WSNs.

The second strategy consists of adjusting the \textit{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

### Table 2

Partial results obtained by applying different aggregate functions

<table>
<thead>
<tr>
<th>Node</th>
<th>Collected data</th>
<th>Aggregate function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td></td>
<td>20(^\circ)–20(^\circ)–21(^\circ)</td>
<td>20(^\circ)</td>
</tr>
<tr>
<td>( B )</td>
<td>21(^\circ)–21(^\circ)–20(^\circ)</td>
<td>21(^\circ)</td>
</tr>
<tr>
<td>Base station</td>
<td>( p_A ) and ( p_B )</td>
<td>20(^\circ)</td>
</tr>
</tbody>
</table>

\( p_A \) is the packet produced by node \( A \) and \( p_B \) is the packet produced by node \( B \).
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.

Fig. 7 shows three possible curves for the function \( h(m) \), each of which having a different value for \( z \) which defines the curved line tendency. For \( z = -0.4 \), 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 \( z = -1.2 \), 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, \( z = -0.6 \), 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. Approximation techniques for producing more precise results

Statistical analysis based on a complete population is the most precise mechanism for drawing conclusions about a phenomenon. However such analysis is not feasible in WSN applications. If performed, the analysis can likely yield unacceptable performance, since it may consume most of the WSN resources. That is because the amount of collected data is often very large. By applying approximation (statistics) techniques one can predict values, which were not actually collected by a sensor, using a small sample – a smaller data set than the population.

In ADAGA, when the sensing interval is informed in a query, it defines a sample that represents the precision of the results as desired by the user. The size (\( z \)) of a given sample can be calculated based on Send interval (\( d \)) and Sense interval (\( g \)) clause values, i.e., \( z \approx \frac{d}{g} \). When \( g \) is increased, a smaller sample with size of \( z' \) is produced, where \( z' \leq z \). Accordingly, less precise results are produced, since a higher measurement error is involved. Thus, the adjustments made by the \( h(m) \) function – the adaptation of the Sense interval – over \( g \) tend to reduce the accuracy of the final result, having therefore a negative effect for the user. In order to avoid that side-effect, ADAGA predicts non-performed detection values by analyzing collected values. Parameters used to compute the number of predictions performed by ADAGA are described in Fig. 8. ADAGA uses those parameters in order to produce results as close as possible to the real results (obtained when no resource constraint is faced).

In order to illustrate how ADAGA predicts non-performed detection values, let us consider the query \( q \) as shown in Fig. 9. Note that sensor nodes should send to the base station packets composed by pairs of the form: distinct temperature value (actually detected) and the corresponding number of detections for each sensor.

Now consider an hypothetical timeline \( T \) representing the period in which data necessary for producing the results for \( q \) (Fig. 9) are collected from the environment.
Discover which are distinct temperature values and the correspondent number of detections in the following delimited geographical region: 3º43’08”S-38º31’51”W and 3º43’16”S-38º31’14”W. The query should be injected into the system just once (on 10-Oct-06 at 14:00:00) and its execution should last 9x10^3 seconds. Data should be collected each 1x10^2 seconds and sent to the base station each 3x10^2 seconds.

- **SCHEDULE**
- **SENSE INTERVAL** 100
- **SEND INTERVAL** 30000
- **TIME WINDOW** 90000

**GROUP BY** `r.regionID`

**WHERE**
- `r.initialLongitude > 383151`
- `r.initialLatitude    > 034308`
- `r. regionID = t. regionID`
- `r.finalLongitude < 383114`
- `r.finalLatitude < 034316`

**AND**
- `t.originID = 1`
- `t.collectedValue >= 120`
- `t.collectedValue < 140`
- `t. collectedValue <= 260`
- `t. collectedValue > 190`

**SELECT** `t.collectedValue, count(t.collectedValue)`

**FROM** `Temperature t, SensorRegion r`
packet \( p_2 \) illustrated in Fig. 12. Observe that in this case, extra memory is required to store estimated detection values and its correspondent number of estimated detections, and thus packet became larger.

(iii) **Weighted average:** a weighted average is computed between the last detected values \( a_i \) and next detected values \( b_j \). In this case a unique pair of values of the form: average value \( e \) and total number of predictions \( k \) is produced. These values are computed as shown in the formulas below

\[
\rho = \sum_{i=1}^{n} k_i, \quad \text{where} \quad 0 < i \leq n, \quad i, n, k_i \in \mathbb{N}
\]

\[
e = \left( \frac{\sum_{i=1}^{n} (a_i + b_i)/2 \cdot k_i}{\rho} \right), \quad \text{where} \quad a_i, b_i \in \mathbb{Q}
\]

When local data are packet, the pair of the form \( e \) and \( k \) is added into the packet. Thus until the 700th second of \( T \), in-memory data would produce packet \( p \) illustrated in Fig. 13. The advantage of using weighted average of successive detected values is that smaller packets are transmitted, since just one prediction value is packed.

(iv) **Mode of detected values:** estimated detections are based on the mode value. The mode of a data sample is the element that occurs most often in the collection. As 10 °C is the most common value \( e \) until the 700th second of \( T \) (see Fig. 10) three predictions \( \rho \) of 10 °C would be computed. It is important to note that when a sample have more than one mode value, some extra memory and processing may be required to store and compute the average of the midpoints in order to produce the final result. Thus until the 700th second of \( T \), in-memory data would produce the following packet (see Fig. 14).

Fig. 15 shows a comparative analysis of the four prediction strategies which can be used by ADAGA. Consider that the base station computes a final average of temperature values based on the distinct temperature values and the correspondent number of detections. Those results were produced by running the query depicted in Fig. 9. When \( g \) is processed by sensor nodes not in resource constraints situation, the average produced is 15.3 °C. When sensor nodes are able to produce the exact result for \( g \), it is required an energy consumption of \( 27.72 \times 10^6 \) nJ.\(^1\)

Fig. 15a compares how close the results produced by the four prediction strategies are to the exact result, considering different resource constraint situations. Observe that the Average of two successive detections strategy stays closer to the exact result value than the other strategies, which means that this strategy produce more precise results. Fig. 15b shows that energy consumption is slightly higher in the Average of two successive detections strategy, since it tends to produce larger amounts of data to be transmitted. Fig. 15c compares the number of performed detections and Fig. 15d presents a comparative analysis of the number of predictions required for each prediction strategy.

The drawback of adjusting sensing activities in accordance with resource constraint situations is that users do not know how approximated received results in the base station are, since initial query parameters may have been

\(^1\) Energy consumption based on a transmission cost of 75 nJ/bit, a reception cost of 50 nJ/bit and a detection and processing cost of 125 nJ/bit.
changed in different sensors. An alternative to solve this problem consists in allowing users to specify the minimum acceptably certainty index (mc), which would be defined by using a new SNQL clause. That clause would inform sensor nodes that just results produced within a minimum certainty index would be acceptable. The certainty index should be computed by each sensor node before a packet sending. In case of mc ≤ (z̄/z), sensor nodes would send local data through the network, otherwise local data would be discarded. Thus, sensor nodes not able to produce results within the acceptable certainty specified in a query would not produce results to answer that query. In this case, sensor nodes are just able to retransmit incoming packets to other nodes.

4.4. 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 (Heinzelman et al., 1999). 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.

In Fig. 16, 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., $20^\circ C - 10$, 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 ($21^\circ C - 10$) with data from $A$, generating a new packet $p_C$. $D$ aggregates its local collected data ($20^\circ C - 10$) 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 Fig. 16).

ADAGA does support packet replication. Nonetheless, ADAGA implements a packet replication mechanism different from that used in flooding strategy for avoiding the problem described in the above paragraph. In 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 $N_0$, 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, two values are stored in its header, the initial number of replicas (copies) of $p$, denoted $c_0$, and the current remaining number of replicas of $p$, $c$. 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 other $c_0 - (m - 1) - \sum_{i=1}^{m}(c_0 - c_i)$ copies are discarded, and $c$ is updated to $c = c_0 - (m - 1) - \sum_{i=1}^{m}(c_0 - c_i)$. 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 ($c = 1$).

In order to illustrate how ADAGA processes in-network aggregation, consider the same example depicted in Fig. 16. Observe in Fig. 17 how data are progressively aggregated until they reach the base station (delays are disregarded). In Fig. 17, 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 \( p_4 \) 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.

4.5. Evaluation

Our experiment was executed on a sensor network simulator, developed in C++, and executed in a Pentium IV machine. This simulator allows the configuration of memory and energy availability. Thus, it is possible to simulate how the ADAGA algorithm would respond in resource constraint situations. We also consider a uniform distribution of temperature values between \(-10^\circ\)C and \(40^\circ\)C. It was considered the query q, depicted in Fig. 9 (see Section 4.5). The base station computes the average temperature based on data received from sensor nodes.

First of all, the q was executed disregarding resource constraints, which produces correct results, according with query parameters specified by the user. The correct result is represented by the dotted line across the graph. Fig. 18a shows how the query result assumes different approximations of the correct result, based on three approaches: (i) proactive, used in ADAGA to monitor energy and memory availability, in order to adjust sensor activities; (ii) reactive, which consists of processing in-network aggregation without monitoring resource usage, thus this approach does not treat resource limitation problems in advance; and (iii) without in-network aggregation, which consists in having sensors just collecting data from the environment and sending them to other nodes. Note that all prediction strategies presented on Section 4.3 are considered proactive, since they were evaluated by running the ADAGA algorithm. The proactive approach represented in Fig. 18 is the Average of two successive detections, which produced the best approximate results.

Note that in 90,000 s, approximately 900 detections would have been made, if memory constraints were disregarded producing the result for the average operation of 15.03 °C. In Fig. 18a, 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. Fig. 18b shows a comparative graph of the amount of collections made. In this case, the proactive approach and the reactive approach have almost the same number of detections, since both perform in-network aggregation. On the other hand, when in-network aggregation is not applied, the number of collections drastically decreases since memory is quickly used up.

Fig. 19 compares energy consumption using those three approaches presented above. Observe that energy consumption increases with memory consumption since more
data are sent to other nodes and thus more power consumption is required. However, the proactive approach achieves a better optimized energy usage when compared to other approaches. When memory availability is enough to compute the query result, all three strategies reach the same power consumption amount.

5. Related work

There is a growing interest in query processing for WSNs. Some research has explored in-network aggregation as an alternative to achieve energy efficiency when propagating data from sensors to sink nodes (Intanagonwiwat et al., 2002; Madden et al., 2002, 2005; Yao and Gehrke, 2003). In-network aggregation approaches are mainly differentiated by their network protocols for routing data. Directed diffusion is proposed in Intanagonwiwat et al. (2000) 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, maintaining an extra cache represents additional overhead. In Intanagonwiwat et al. (2002) 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 (Madden et al., 2002) 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 4.4, 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 (Madden et al., 2005), which has some simple extensions to SQL for controlling data acquisition. Madden et al. show how acquisitional issues influence query optimization, dissemination, and execution (Madden et al., 2005). 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 have proposed an adaptive in-network aggregation operator, called ADAGA, for query processing in sensor nodes of a WSN. ADAGA is able to adjust sensor activities according to energy consumption and memory usage, in order to maximize sensor lifetime (by saving energy) and also the accuracy of query results. Besides performing in-network aggregation in WSNs with packet replication, ADAGA predicts non-performed detection values by applying statistical techniques, such as weighted average. The results obtained through experiments prove the efficiency of ADAGA w.r.t power consumption and result precision. Furthermore, we briefly described a query language for sensor networks, called SNQL, which is especially designed to provide initial (input) parameters (e.g., data sensing and data sending intervals) to be used by ADAGA.

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


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


Madden, Samuel R., Franklin, Michael J., Hellerstein, Joseph M., Hong, W., 2002. TAG: a tiny aggregation service for ad hoc sensor networks. OSDI.


Angelo Brayner received the M.Sc. degree in Computer Science from the State University of Campinas (UNICAMP), Brazil, in 1994. He received the Ph.D. degree from the University of Kaiserslautern, Germany. He has been with the University of Fortaleza, Brazil, since 2001 as full professor and leader of the CEARA (advanCEd dAtabase Research) research group. His current research interests include query processing in wireless sensor networks and mobile databases and high performance transaction systems.

Ronaldo Menezes is an associate professor in computer science at Florida Institute of Technology. During his M.Sc. at the State University of Campinas (UNICAMP), Brazil, he worked on Models of Parallel Computing, finishing the degree in 1995. He was awarded his Ph.D. in computer science by the University of York, UK, in the field of Coordination Systems in 2000. His current research interests include coordination of multi-agent systems, and scalability of distributed systems and mobile ad-hoc networks (MANETs). Most recently, he has focused his research efforts on the usefulness of biological metaphors based on insect-behavior (aka Swarms) in various fields including homeland security, software engineering as well as coordination system.

aretusa Lopes obtained her M.Sc. degree in Computer Science at University of Fortaleza with focus in the Database research area (2006). She has a specialist certificate in Networks and Internet Technology (2003) and a B.Sc. in Computer Science (1999), both at University of Fortaleza, Brazil. Between 1997 and 1999 she was part of a team in a research project with focus in Internet and Database technology sponsored by CNPq – The Brazilian National Council for Scientific and Technological Development. She is currently working as a Systems Analyst at University of Alberta, Canada. Previously, she worked as a Database Administrator at SERPRO – The Brazilian Federal Data Processing Service (2004 to 2007) and as a Systems Analyst at Edson Queiroz Group, Brazil (2000 to 2004).

Diorgens Miguel Meira was born in Brumado, Bahia (Brazil) in 1964, April-07. He has Bachelor degree in Computer Science from the University of Fortaleza (1993). He is currently master degree student, participating of database research group in UNIFOR. Diorgens works in Brazilian Northeast Bank (BNB), where, from 1984 to 2003, he was member of teams of development of basic software and data communication tools. He is currently an auditor of technology information in BNB.

ricardo Vasconcelos has Bachelor degree in Computer Science from the University of Fortaleza (2006). In 2006 he joined to the Research Group chaired by Prof. Brayner. Since then, he is working for receiving the M.Sc. degree from the University of Fortaleza. His current research interests include query processing in wireless sensor networks.