Available online at www.sciencedirect.com
ScienceDirect
Information Systems 31 (2006) 770–792
www.elsevier.com/locate/infosys

Processing approximate aggregate queries in wireless sensor networks☆

Antonios Deligiannakis[a], Yannis Kotidis[b,*], Nick Roussopoulos[a]

[a]University of Maryland, USA
[b]Department of Information Systems and Analysis, AT&T Labs-Research, P.O. Box 971, Florham Park, NJ 07932-0971, USA

Received 5 February 2004; received in revised form 1 February 2005; accepted 8 February 2005

Abstract

In-network data aggregation has been recently proposed as an effective means to reduce the number of messages exchanged in wireless sensor networks. Nodes of the network form an aggregation tree, in which parent nodes aggregate the values received from their children and propagate the result to their own parents. However, this schema provides little flexibility for the end-user to control the operation of the nodes in a data sensitive manner. For large sensor networks with severe energy constraints, the reduction (in the number of messages exchanged) obtained through the aggregation tree might not be sufficient. In this paper, we present new algorithms for obtaining approximate aggregate statistics from large sensor networks. The user specifies the maximum error that he is willing to tolerate and, in turn, our algorithms program the nodes in a way that seeks to minimize the number of messages exchanged in the network, while always guaranteeing that the produced estimate lies within the specified error from the exact answer. A key ingredient to our framework is the notion of the residual mode of operation that is used to eliminate messages from sibling nodes when their cumulative change to the computed aggregate is small. We introduce two new algorithms, based on potential gains, which adaptively redistribute the error thresholds to those nodes that benefit the most and try to minimize the total number of transmitted messages in the network. Our techniques significantly reduce the number of messages, often by a factor of 10 for a modest 2% relative error bound, and consistently outperform previous techniques for computing approximate aggregates, which we have adapted for sensor networks.

Keywords: Sensor networks; Aggregate queries; Approximation

1. Introduction

Densely distributed sensor networks are used in a variety of monitoring applications ranging from measurements of meteorological data (like temperature, pressure, humidity), noise levels,

☆Recommended by Bettina Kemme, School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building, Room 318, Montreal, Quebec, Canada H3A 2A7.

*Corresponding author. Tel.: +1 973 360 8347; fax: +1 973 360 8077.
E-mail address: kotidis@research.att.com (Y. Kotidis).

0306-4379/$ - see front matter
doi:10.1016/j.is.2005.02.001

chemicals etc., to complex military vehicle surveillance and tracking applications. Real time (or near-real time) measurements taken from biological and chemical sensor networks are also used in conjunction with modeling and data mining tools in large environmental databases for evaluating environmental conditions and security decision making [1].

A common characteristic of sensor node applications revolves around the severe energy and bandwidth constraints that are met in such networks. In many applications sensor nodes are powered by batteries and replacing them is not only very expensive, but often impossible. For example, sensor nodes thrown in a disaster area need to operate *unattended* within an uncontrollable environment. Thus, energy-aware protocols involving the operation of the nodes are required to ensure the longevity of the network [2,3]. The bandwidth constraints arise from the wireless nature of the communication among the nodes, the short ranges of their radio transmitters and the high density of network nodes in some areas. Energy and bandwidth consumption in sensor networks are strongly correlated, since radio transmission is the most important source of energy drain on sensor nodes [3,4].

Designing efficient data dissemination protocols is, thus, essential for the survivability of large scale sensor networks. Furthermore, the abundance of data that can be collected in networks consisting of thousand of nodes might be overwhelming for the end-user to process. Aggregation is an effective means to reduce the data measurements into a small set of comprehensive statistics, like sum, min, max, average, etc. At the same time, aggregation, when performed *inside the network*, can substantially reduce the amount of transmitted data [2,5,15,7]. At the core of these techniques lies the notion of an aggregation tree that provides the conduit within which detailed measurements taken from the sensors are aggregated on their route to the monitoring node. Non-leaf nodes of that tree aggregate the values of their children before transmitting the aggregate result to their parents. In [2], after the aggregation tree has been created, the nodes carefully schedule the periods when they transmit and receive data. The idea is for a parent node to be listening for values from its child nodes within specific intervals of each *epoch* (the user specified period between updates to the query result), and vice versa. This allows the nodes to power-down their radios when not necessary and, thus, reduce energy consumption. At each epoch, ideally, a parent node coalesces all partial aggregates from its child nodes and transmits upwards a single partial aggregate for the whole subtree.

All the above techniques try to limit the number of transmitted data while always providing accurate answers to posed queries. However, there are many instances where the application is willing to tolerate a specified error in order to reduce the bandwidth consumption and increase the lifetime of the network. In [8], Olston et al. study the problem of error-tolerant applications where the users register continuous queries along with strict precision constraints at a central *stream processor*. The stream processor then dynamically distributes the error budget to the remote data sources by installing filters on them that necessitate the transmission of a data value from each source only when the source's observed value deviates from its previously transmitted value by more that a threshold specified by the filter.

As we will demonstrate in this paper, the algorithms in [8] cannot be directly applied to monitoring applications over sensor networks. While the nodes in sensor networks form an aggregation tree where messages are aggregated and, therefore, the number of transmitted messages depends on the tree topology, [8] assumes a flat setup of the remote data sources, where the cost of transmitting a message from each source is independent to what happens at the other data sources. Moreover, as we will show in this paper, the algorithms in [8] may exhibit several undesirable characteristics for sensor networks, the most important of which are:

- The existence of a few *volatile* sensor nodes, that is nodes that exhibit large variance in their measurements, will make the stream processor distribute much of its available budget to these nodes, without any significant benefit and at the expense of all the other sensor nodes.

- The error distribution assumes a worst-case behavior of the sensor nodes. If any node exceeds its specified threshold, then its data needs to be propagated to the monitoring node. However, there might be many cases when changes from different data sources effectively cancel out each other. When this happens frequently, our algorithms should exploit this fact and, therefore, prevent unnecessary messages from being propagated all the way to the root node of the aggregation tree.

In this paper, we develop new techniques for in-network data aggregation, when the monitoring application is willing to tolerate a specified error threshold. Our techniques operate by considering the potential benefit of increasing the error threshold at a sensor node, which is equivalent to the amount of messages that we expect to save by installing a larger filter at the node. The result of using this gain-based approach is a robust algorithm that is able to quickly identify volatile data sources and eliminate them from consideration. Moreover, we introduce the *residual mode of operation*, during which a parent node may eliminate messages from its children nodes in the aggregation tree when the cumulative change from these sensor nodes is small. Finally, unlike the algorithms in [8], our algorithms operate with only local knowledge, where each node simply considers statistics from its children nodes in the aggregation tree. This allows for more flexibility in designing adaptive algorithms and is a more realistic assumption for sensors nodes with very limited capabilities [2].

Our contributions are summarized as follows:

(1) We present a detailed analysis of the current protocols for in-network data aggregation in the case of error-tolerant applications, along with their shortcomings.
(2) We introduce the notion of the *residual* mode of operation. In cases when the cumulative change in the observed quantities of multiple sensor nodes is small, this operation mode helps filter out messages close to the sensors and prevents these messages from being propagated all the way to the root of the aggregation tree.
(3) We introduce the notion of the *potential gain* of a node or an entire subtree and employ it as an indicator of the benefit of increasing the error thresholds in some nodes of the subtree. We then present two adaptive algorithms that dynamically determine how to rearrange the error thresholds in the aggregation tree using simple, local statistics on the potential gains of the nodes. Unlike previous techniques, where nodes are treated independently, our algorithms take into account the tree hierarchy and the resulting interactions among the nodes. The difference between our two proposed algorithms is that the first one redistributes the error budget in a top-down manner, starting from the *Root* node of the tree, while the second one uses a more localized approach, redistributing the budget among parent–child nodes at each level of the tree.
(4) We present an extensive experimental analysis of our algorithms. Our experiments demonstrate that, for the same maximum error threshold of the application, our techniques have a profound effect on reducing the number of messages exchanged in the network and outperform previous algorithms, which we have adapted for sensor networks.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3 we provide an introduction to sensor nodes and the data aggregation process. In Section 4 we describe the algorithms presented in [8], along with their shortcomings when applied to sensor networks. Section 5 presents our extensions and algorithms for dynamically adjusting the error thresholds of the sensor nodes. Section 6 contains our experiments, while Section 7 contains concluding remarks and discusses future work.

2. Related work

The development of powerful and inexpensive sensors in recent years has spurred a flurry of research in the area of sensor networks, with particular emphasis in the topics of network self-configuration [9], data discovery [4,10], distributed data storage [11–13], energy efficient data routing [14,15] and in-network query processing [2,5,7,16]. A survey of the applications and the challenges that sensor networks introduce is presented in [4].

For monitoring queries that aggregate the observed values from a group of sensor nodes, [5] suggested the construction of a greedy aggregation tree that seeks to maximize the number of aggregated messages and minimize the amount of the transmitted data. To accomplish this, nodes may delay sending replies to a posed query in anticipation of replies from other queried nodes. A similar approach is followed in the TAG [2], TinyDB [3] and Cougar [7] systems. In [17], a framework for compensating for packet loss and node failures during query evaluation is proposed. In [11], additional issues such as selecting the optimal aggregation tree given a query workload and optimizing the scheduling of the transmissions to minimize collisions are discussed.

The work in [2] also addressed issues such as query dissemination, sensor synchronization to reduce the amount of time a sensor is active and, therefore, increase its expected lifetime, and also techniques for optimizations based on characteristics of the used aggregate function. Similar issues are addressed in [3], but the emphasis is on reducing the power consumption by determining appropriate sampling rates for each data source. The above work complements ours in many cases, but our optimization methods are driven by the error bounds of the application at hand.

Our work is also related to the area of continuous queries over data streams, which has been broadly studied in recent years [18–20]. Olston et al. in [21–23] investigated the tradeoffs between precision and performance in cached and replicated data. More recently, in [8] the issue of applications that may tolerate a specified error threshold was discussed and a novel dynamic algorithm for minimizing the number of transmitted messages was suggested. While our work shares a similar motivation with the work in [8], our methods apply over a hierarchical topology, such as the ones that are typically met in continuous queries over sensor networks. Similarly, earlier work in distributed constraint checking [24,25] cannot be directly applied in our setting, because of the different communication model and the limited resources at the sensors. The work of [6] provides quality guarantees during in-network aggregation, like our framework, but this is achieved through a uniform allocation strategy and does not make use of the residual mode of operation that we introduce in this paper. The evaluation of probabilistic queries over imprecise data was studied in [26]. Extending this work to hierarchical topologies, such as the ones studied in our paper, is an open research topic.

3. In-network data aggregation

Depending on their application, sensor nodes typically operate under one of two possible modes. In the *batch* processing mode sensor nodes collect data until either the collected data reaches a specified size, or until a maximum amount of time since the last transmission has elapsed. The collected data is then processed locally and periodically forwarded to a base station for further processing and analysis. In the *data-driven* mode, the time of the data transmission is determined by the values of the collected data. For example, a node may be programmed to transmit its measurement when a collected value deviates by more than 20% from its previously transmitted value. The applications that we consider in this paper involve the data-driven processing mode of sensor nodes. For the batch processing mode, data reduction techniques such as the ones presented in [27] may be applied.

3.1. Data aggregation process

We now briefly describe the data aggregation process in sensor networks when 100% accuracy (ignoring network delays or lost messages) is desired in the querying node, and when utilizing the TAG [2] model to synchronize the transmission of data values by the sensor nodes.

Consider a node *Root*, which initiates a continuous query over the values observed by a set of data sources, and requests that the results of this query be reported to it at regular time periods. The

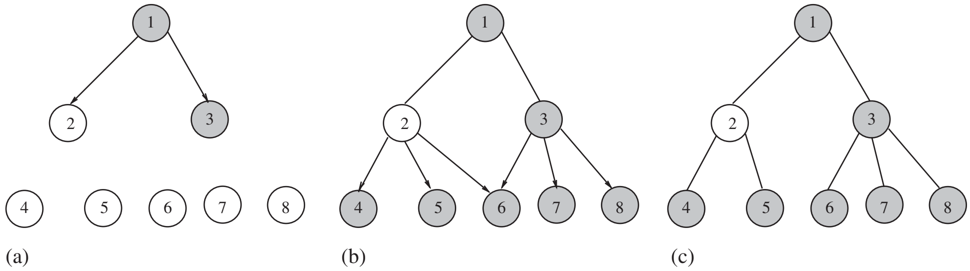

Fig. 1. Query dissemination process (steps (a) and (b)) and formed aggregation tree (step (c)).

time interval between two such consecutive time periods is referred to as the *epoch* of the query. The continuous query is disseminated through the network in search of the sensor nodes that collect data relevant to the posed query. While each such node may have received the announcement of the query through multiple nodes, it only selects one of these nodes as its *parent* node, through which it will propagate its results towards the *Root* node. The flow of the query results forms a tree, rooted at the *Root* node, which is commonly known as the *aggregation tree* [2,4,5]. The query dissemination process and a sample aggregation tree are depicted in Fig. 1. The nodes in the aggregation tree can be classified as either *active* or *passive*. Active nodes (marked gray in the figure) collect measurements relevant to the query, while passive nodes (marked white in the figure) simply facilitate the propagation of results towards the *Root* node.

At each epoch, each sensor node N_i calculates the partial aggregate corresponding to the query result produced by measurements obtained by sensor nodes in the subtree of N_i. This calculation is performed bottom-up, where each node first waits to receive any updated partial aggregate values from its children nodes (in the aggregation tree) and then combines these values with its own collected measurements (if this is an *active* node) to produce the partial aggregate for its subtree.

3.2. Challenges and opportunities during in-network data aggregation

We now discuss some challenging characteristics of in-network data aggregation that motivate our techniques.

3.2.1. Hierarchical structure of nodes

The hierarchical organization of the nodes results in a single aggregate value transmitted by each node towards its parent in the aggregation tree. However, not all kinds of information relevant to the query execution process can be aggregated in the same manner. Consider, for example, a scenario where only a non-predetermined subset of the sensor nodes in the aggregation tree makes a transmission within each epoch. This scenario is typical, as we will discuss later in this paper, in the evaluation of approximate aggregate continuous queries. If some application requires that the *Root* node know exactly which nodes made a transmission during each epoch, then each transmitting node needs to piggyback its identifier (*id*) to each message that it transmits. Note that, because messages are combined in the aggregation tree, in this scenario each message transmitted by a node N_i will contain the node *ids* of all the transmitting nodes in the subtree of N_i. Obviously, this side information can potentially be excessive; it may not even fit within the maximum packet size, thus requiring that it be fragmented and transmitted through multiple messages.

A similar problem occurs whenever a node requires *individual* statistics from the sensors in the aggregation tree. This information cannot be aggregated, since each individual node statistic needs to be accompanied by the node's identifier. Thus, any technique or algorithm that requires individual node statistics will result in the transmission of large amounts of information, which may outweigh the benefits of in-network aggregation.

3.2.2. Nodes with different characteristics

In a large sensor network, nodes with widely different characteristics may exist. The measurements of some nodes may be either significantly higher or exhibit much larger variance than the measurements of some other nodes. For example, in an application where sensors are used to trace moving objects within their vicinity, some sensor nodes may detect a large number of moving objects, while others may detect only few, if any. Moreover, the number of detected moving objects over time by each sensor may change either rapidly, if the speed of the objects is significant, or very slowly, if the objects are moving slowly. Throughout this paper, we refer to sensor nodes that exhibit large variance in their measurements as *volatile* nodes. Proper handling of volatile nodes is crucial, as an ill-designed algorithm may allocate a lot of resources to them at the expense of other nodes in the network.

3.2.3. Negative correlations in neighboring areas

During the data aggregation process, each node calculates the partial aggregate value of its subtree and forwards this new value to its parent node in the aggregation tree. However, there might be cases when changes from nodes belonging to different subtrees of the aggregation tree either cancel out each other, or result in a very small change in the value of the calculated aggregate. This may happen either because of a random behavior of the data, or because of some properties of the measured quantity.

Consider for example the aggregation tree of Fig. 1(c), and assume that each node observes the number of items moving within the area that it monitors. If some objects move from the area of node 4 to the area of node 5, then the changes that will be propagated to node 2 will cancel out each other. In this case, the partial aggregate value calculated by node 2 does not change and, therefore, there is no need for node 2 to make a transmission. Node 1 may then safely assume that the partial aggregate value of node 2 has not been modified. Even when the overall change of a node's aggregate value is non-zero, but reasonably small, the filtering of transmissions from this node may result in a large number of saved messages with only minimum effect in the reported aggregate result. In an *approximate* data aggregation application it is crucial to detect and exploit areas where such *negative correlations* occur frequently.

4. Existing techniques and their drawbacks

In this section, we will demonstrate that straightforward extensions to the algorithm of [8] for sensor network applications result in several shortcomings due to the issues discussed in Section 3.2. The original algorithm of [8] was devised for applications containing a non-hierarchical node setup, where all the nodes in the aggregation tree can be assumed to be direct children of the *Root* node that initiates the query, and therefore, all the messages are aggregated only on that node. Moreover, due to the node setup considered in [8], all the nodes in the aggregation tree collect data relevant to the query (*passive* nodes do not exist).

In our discussion hereafter, we will use the term *burden-based adjustment* (*BBA*) to refer to the adaptation of the algorithm of [8] for approximate in-network data aggregation, combined with the model of TAG [2], with the latter being used in order to coalesce messages within the aggregation tree.

4.1. Burden-based adjustment of node filters

Consider a node *Root*, which initiates a continuous query over the values observed by a set of data sources. This continuous query aggregates values observed by the data sources, and produces a single aggregate result. For each defined query, a maximum error threshold, or equivalently a precision constraint *E_Global* that the application is willing to tolerate is specified. The algorithm will install filters at each queried data source, that will help limit the number of transmitted messages from the data source. The selection process for the filters enforces that at any moment after the installation of the query to the data sources, the aggregate value reported at node *Root* will lie within the specified error threshold from the true aggregate value (ignoring network delays or lost messages).

Initially, a filter F_i is installed in every data source S_i. Each filter F_i is an interval of real values $[L_i, H_i]$ of width $W_i = H_i - L_i$, such that any source S_i whose current observed value $Current_i$ lies outside its filter F_i will need to transmit its newly calculated partial aggregate value, while also taking into account any messages from its children nodes, towards the *Root* node and then re-center its filter around this transmitted value, by setting $L_i = Current_i - W_i/2$ and $H_i = Current_i + W_i/2$. If $Current_i$ lies within the interval specified by the filter F_i, then this value does not need to be transmitted. Note, however, that for any non-leaf node in the aggregation tree, any messages that it receives from its children (unless the resulting aggregate change from these messages is zero) need to be propagated towards the *Root* node, since the node's filter is applied only to the node's observed data value and not on the partial aggregate of its subtree.[1] In this case, the node may include for free in the newly calculated partial aggregate its current observed value and recenter its filter around this value. It is important to emphasize that the initial error guarantees should not be violated by the filter initialization. For example, for the *SUM* aggregate function the following inequality must be true: $\sum_i W_i/2 \leqslant E\_Global$.

In order for the algorithm to be able to adapt to changes in the characteristics of the data sources, the widths W_i of the filters are periodically adjusted. Every *Upd* time units, *Upd* being the *adjustment period*, each filter shrinks its width by a *shrink percentage* (*shrinkFactor*). At this point, the *Root* node obtains an *error budget* equal to $(1 - shrinkFactor) \times E\_Global$, which it can then distribute to the data sources. The decision of which data sources will increase their window W_i is based on the calculation of a *Burden Score* metric B_i for each data source, which is defined as $B_i = C_i/(P_i \times W_i)$. In this formula, C_i is the cost of sending a value from the data source S_i to the *Root* and P_i is the *estimated streamed update period*, defined as the estimated amount of time between consecutive transmissions for S_i over the last period *Upd*. For a single query over the data sources, it is shown in [8] that the goal would be to try and have all the burden scores be equal. Thus, the *Root* node selects the data sources with the largest deviation from the target burden score (these are the ones with the largest burden scores in the case of a single query) and sends them messages to increase the width of their windows by a given amount. The process is repeated every *Upd* epochs.

4.2. Drawbacks of the BBA algorithm

We now discuss some of the key drawbacks of the *BBA* algorithm when applied to sensor network applications. Our discussion will be based on the data aggregation characteristics discussed in Section 3.2.

4.2.1. Hierarchical structure of nodes

In order to calculate the burden score of each sensor N_i, the *Root* node needs to estimate the node's *estimated streamed update period* P_i, and the cost C_i of node N_i transmitting values towards the *Root* node. In order for the *Root* to estimate the P_is, each node either needs to transmit at the last epoch of the update period the number of total transmissions that it performed, or piggyback in each message that it transmits its identifier. Obviously, this amount of side information needed is excessive (see Section 3.2.1) and may outweigh the benefits of approximate data aggregation. Note that in a non-hierarchical setup of nodes, this problem would not occur, since the *Root* node would be able to identify from any received packet's header the sender of the message, and accurately compute the number of transmissions by each node.

Calculating the average cost of the transmissions made by a node is more complex. In a non-hierarchical setup of the nodes, this cost could depend on parameters like the bandwidth capacity of the link between each node and the *Root* and could be considered to be either fixed throughout the query execution, or change only occasionally. In a hierarchical setup, this quantity, if measured

[1] In the experiments we also investigate the option of applying the error filter to the partial aggregate value of the subtree. However, this modification typically resulted in more transmitted messages than the presented one.

in the number of messages resulting from each node's transmission, depends on the topology of the other transmitting nodes in the aggregation tree. This point can be more easily understood with an example. Consider the two scenarios depicted in Fig. 2. In both scenarios, only two nodes make a transmission (the transmitted messages are depicted by bold, thick arrows). However, the transmitted messages are aggregated at different nodes of the aggregation tree. In the first scenario (Fig. 2(a)), nodes 4 and 6 make a transmission, and nodes 2 and 3 propagate these messages towards the *Root* node. In this case, each transmission from nodes 4 and 6 is responsible for generating 2 messages. On the other hand, in the second scenario (Fig. 2(b)), nodes 4 and 5 make a transmission, and node 2 propagates a single message to node 1. Therefore, each transmission is responsible for only 3/2 messages in this case.

To calculate the actual cost C_i (in number of generated messages) for each node transmission (or an average cost over multiple transmissions), the *Root* node requires knowledge of not only which nodes made a transmission within each epoch, but also of the exact topology (parent–child relationships) of these nodes and, furthermore, whether these nodes made a transmission because their monitored value laid outside the node's filter, or simply made a transmission to forward changes in their calculated partial aggregate because of transmissions by some of their descendants. However, this is a completely unrealistic scenario, since too much information would need to be communicated, namely the exact topology and the root-cause of each transmission. Therefore, the techniques introduced in [8] can be applied in our case only by using a heuristic function to estimate C_i. In Section 6 we describe such a heuristic.

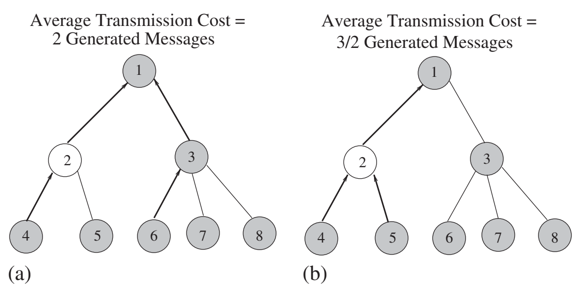

Fig. 2. Two transmissions scenarios with different costs for each transmission.

4.2.2. *Nodes with different characteristics*

One of the principle ideas behind the adaptive algorithms presented in [8] is that an increase in the width of a filter installed in a node will result in a decrease at the number of transmitted messages by that node. While this is an intuitive idea, there are many cases, even when the underlying distribution of the observed quantity does not change, where an increase in the width of the filter does not have any impact in the number of transmitted messages. To illustrate this, consider a node whose values follow a random step pattern, meaning that the observed value at each epoch differs by the observed value in the previous epoch by either $+\Delta$ or $-\Delta$. In this case, any filter with a window whose width is less than $2 \times \Delta$ will not be able to reduce the number of transmitted messages. A similar behavior may be observed in cases where the measured quantity exhibits a large variance. In such cases, even a filter with considerable width may not be able to reduce but a few, if any, transmissions.

The main reason why this occurs in the *BBA* algorithm is because the burden score metric being used does not give any indication about the expected benefit that we will achieve by increasing the width of the installed filter at a node. In this way, a significant amount of the maximum error budget that the application is willing to tolerate may be spent on a few nodes whose measurements exhibit the aforementioned volatile behavior (note that due to the large number of their transmissions these nodes will also exhibit large burden scores), without any real benefit.

4.2.3. *Negative correlations in neighboring areas*

According to the algorithms in [8], each time the value of a measured quantity at a node N_i lies outside the interval specified by the filter installed at N_i, then the new calculated partial aggregate value of the node is transmitted and propagated to the *Root* node. In this case, negative correlations, such as the ones described in Section 3.2.3 are not exploited and messages cannot be prevented from

Table 1
Symbols used in our algorithms

Symbol	Description
N_i	Sensor node i
W_i	The width of the filter of sensor N_i
$E_i = W_i/2$	Maximum permitted error in node N_i
E_Sub_i	Maximum permitted error in entire subtree of node N_i
E_Global	Maximum permitted error of the application
V_Cur	The latest measurement obtained by the node (if active)
Upd	Update period of adjusting error filters
shrinkFactor	Shrinking factor of filter widths
T	Number of nodes in the aggregation tree
Root	The node initiating the continuous query
Gain	The estimated gain of allocating additional error to the node
CumGain	The estimated gain of allocating additional error to the node's entire subtree
CumGain_Sub[*i*]	The estimated gain of allocating additional error to the node's *i*-th subtree

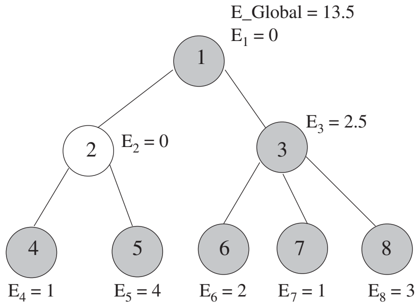

Fig. 3. Sample aggregation tree.

reaching the *Root* node. Even when we modify the *BBA* algorithm to take into account negative correlations (see Section 6), performance is often worse, because *BBA* cannot distinguish on the true cause of a transmission (change on local measurement or change in the subtree).

5. Our algorithms

In this section, we first provide a high-level description of our framework and then present the details of our algorithms for dynamically modifying the widths of the filters installed in the sensor nodes. The notation that we will use in the description of our algorithms is presented in Table 1.

5.1. A new framework for approximate in-network data aggregation

We assume that the aggregation tree (e.g. Fig. 3) for computing and propagating the aggregate has already been established. Techniques for discovering and modifying the aggregation tree are illustrated in [2]. Our algorithms will install a filter at each node N_i in the aggregation tree, independently on whether the node is an active or passive one. This is a distinct difference from the framework of [8], where filters are assigned only to active nodes.

In our discussion we focus on queries containing the *SUM* aggregate function. The *COUNT* function can always be computed exactly as the number of active nodes in the aggregation tree, while the *AVG* function can be computed by the *SUM* and *COUNT* aggregates. As the work in [8] demonstrated, adaptive filter adjustment algorithms for the *MAX* and *MIN* aggregate functions make sense only when considering a multi-query optimization scenario.

Fig. 3 shows the maximum error of each filter for a query calculating the *SUM* aggregate over the active nodes of the tree.[2] Notice that the sum of the errors specified is equal to the maximum error that the application is willing to accept (*E_Global*). Moreover, there is no point in placing an error filter in the *Root* node, since this is where the result of the query is being collected. This can change, when the *Root* node collects and transmits the aggregate to a distant base station. The modifications to all algorithms considered here are straightforward.

We now describe the protocol of propagating values in the aggregation tree assuming the TAG model [2] is used to synchronize the transmissions

[2] The width of the error filter in node 2 may in general be non-zero in our algorithms.

between parent and children nodes in the aggregation tree:

- An active leaf node i obtains a new measurement and forwards it to its parent if the new measurement lies outside the interval $[L_i, H_i]$ specified by its filter.
- A passive (non-leaf) node awaits for messages from its children. If one or more messages are received, they are combined and forwarded to its own parent only if the new partial aggregate value of the node's subtree does not lie within the interval specified by the node's filter. Otherwise, the node remains idle.
- An active non-leaf node obtains a new measurement and waits for messages from its children nodes as specified in [2]. The node then recomputes the partial aggregate on its subtree (which is the aggregation of its own measurement with the values received by its child-nodes) and forwards it to its parent only if the new partial aggregate lies outside the interval specified by the node's filter.

Along this process, the value sent from a node to its parent is either (i) the node's measurement if the node is a leaf or (ii) the partial aggregate of all measurements in the node's subtree (including itself) if the node is an intermediate node. In both cases, a node remains idle during an epoch if the newly calculated partial aggregate value lies within the interval $[L_i, H_i]$ specified by the node's filter. This is a distinct difference from [8], where the error filters are applied to the values of the data sources, and not on the partial aggregates calculated by each node.

5.2. Operation of nodes

The operation of each sensor node is described in Algorithm 1 (notation from Table 1). The algorithms consists of four major tasks: *initialization*, *adjustment of filters*, *aggregation* and *transmission of new aggregate*. These tasks are discussed in detail below.

Initialization (Lines 1–3): A filter is initially installed in each node of the aggregation tree, except for the *Root* node (Line 1). The initial width of each filter is important only for the initial stages of the network's operation, as our dynamic algorithm will later adjust the sizes of the filters appropriately. In our experiments we initialize the widths of the error filters similarly to the *uniform allocation* method. For example, in the case when the aggregate function is the function *SUM* and there are N_{active} active nodes in the aggregation tree (excluding the *Root* node) then each active node is assigned the same fraction E_Global/N_{active} of the error *E_Global* that the application is willing to tolerate.

We note that *E_i* (Line 1) is the maximum permitted error in node N_i, while E_Sub_i is the maximum permitted error in the entire subtree of node N_i. Thus, for the *SUM* function, E_Sub_i is the sum of *E_i* and all *E_j*, where N_j is a descendant of node N_i in the aggregation tree.

Adjustment of filters (Lines 5–10): This adjustment phase is performed every *Upd* epochs. The first step is for all nodes to shrink the widths of their filters by a shrinking factor *shrinkFactor* ($0 < shrinkFactor \leqslant 1$). After this process, the *Root* node has an error budget of size $E\_Global \times (1 - shrinkFactor)$, where *E_Global* is the maximum error of the application, that it can redistribute recursively to the nodes of the network (Lines 8–10). This redistribution process is done using a statistic called the *cumulative gain* of the node, which is a single value and is the only statistic propagated to the parent node at each transmission. Details of the adjustment process will be given later in this section.s At each epoch the node also updates some statistics (Line 21), which will be later used to adjust the widths of the filters.

Algorithm 1. Operation of nodes

Input: *E_Sub* {The maximum permitted error for the subtree of this node}
{*E* is the total maximum permitted error of the node itself}
{*V_Self* is the value of the node's measured quantity at its last transmission}

{*LastReceived*[*i*] is the last received partial aggregate value of the node's *i*-th subtree}
1:In each node initialize E_i using uniform allocation policy and calculate E_Sub_i
2:$NewAggr = 0$ {Current partial aggregate}
3:$LTA = 0$ {Last transmitted partial aggregate}
{Every *Upd* epochs the widths of the filters will shrink.}
4:**for** each epoch *ep* **do**
5: **if** $ep > 0$ AND *ep* modulo $Upd = 0$ **then**
6: $E\_Sub = shrinkFactor * E\_Sub$ $\{0 \leqslant shrinkFactor < 1\}$
7: $E = shrinkFactor * E$
8: **if** received message from father to increase error of subtree by E_Additional **then**
9: $E\_Sub += E\_Additional$
10: Distribute *E_Additional* to self and subtrees and clear all gain related statistics
11: **if** node is active **then**
12: Get current measurement *V_Curr*
13: Wait for messages from children nodes.
14: $\Delta\text{ChAggr} = 0$
15: **for** Each Child *i* **do**
16: **if** Child *i* transmitted an aggregate value V_i and its cumulative gain $CumGain_i$ **then**
17: $\Delta ChAggr \mathrel{+}= V_i - LastReceived[i]$ {Needed for non-residual operation}
18: $LastReceived[i] = V_i$
19: $CumGain\_Sub[i] = CumGain_i$ {Store the cumulative gain of the node's subtrees}
20: $NewAggr = Combine(LastReceived, V\_Curr)$
21: $(Gain, CumGain) = \text{UpdateExpectedGain}(NewAggr, LTA, E, E\_Sub, Gain, CumGain\_Sub)$
22: **if** (nonResidualOperation **AND** $((\Delta ChAggr > 0)$ **OR** $|V\_Self - V\_Curr| > E))$ **OR** (ResidualOperation **AND** $|NewAggr - LTA| > E$) **then**
23: $V\_Self = V\_Curr$
24: $LTA = NewAggr$
25: Transmit (*NewAggr*, *CumGain*) to parent node and re-center the error filter

Aggregation (Lines 11–20): In each epoch, the node obtains a measurement related to the observed quantity if it is an active node (Lines 11–12), and then waits for messages from its children nodes containing updates to their measured aggregate values (Line 13). We here note that each node computes a partial aggregate based on the values reported by its children nodes in the tree. This is a recursive procedure which ultimately results in the evaluation of the aggregate query at the *Root* node. After waiting for messages from its children nodes, the current node computes the new value of the partial aggregate based on the most current partial aggregate values it has received from its children (Line 20). Variable *LastReceived*[*i*] stores the last received partial aggregate value of the root of node's *i* subtree (Line 18).

Aggregation is performed through a call to the *Combine* function. The specific implementation depends on the aggregate function specified by the query. In Table 2 we provide its implementation for the most common aggregate functions. In the case of the *AVG* aggregate function, we calculate the sum of the values observed at the active nodes, and then the *Root* node will divide this value with the number of active nodes participating in the query.

Transmission of new aggregate (Lines 22–25): After calculating the current partial aggregate, the node must decide whether it needs to transmit a measurement to its parent node or not. This depends on the operation mode being used. In a *non-residual* mode, the node would have to transmit a message either when the value of the measured quantity at the node itself lies outside its filter, or when at least one of the subtrees has transmitted a message and the new changes do not exactly cancel out each other ($\Delta ChAggr > 0$). This happens because in the *non-residual* mode (e.g. the original algorithm of [8]) the error filters are applied to the values measured by each node, and not to the partial aggregates of the subtree. On the contrary, in a *residual* mode of operation, which is the mode used in our algorithms, the node transmits a message only when the value of the new partial aggregate lies outside the node's filter. In both modes of operation the algorithm that distributes the available error enforces that for any node N_i, its calculated partial aggregate will never deviate by more than E_Sub_i from the actual partial aggregate of its subtree (ignoring propagation delays and lost messages). When a node makes a transmission, it caches its current state that includes its latest measurement V_Curr (which is copied to variable V_Self).

Table 2
Definition of the *Combine* function

Aggregate	Implementation of *Combine* function
SUM/AVG	$V\_Curr + \sum_i LastReceived[i]$
MAX	$\max\{V\_Curr, \max_i\{LastReceived[i]\}\}$
MIN	$\min\{V\_Curr, \min_i\{LastReceived[i]\}\}$

Consider the aggregation tree of Fig. 3. Assume that the posed query involves the sum of values in the active nodes of the tree (all nodes except for node 2), and that the maximum error that the application is willing to tolerate is 13.5, as shown in Fig. 3. We will explain in detail the transmission of messages for the residual mode of operation, for the sample error filters shown in the figure.

In Table 3 we present an example based on the aggregation tree of Fig. 3. In this table we show the current observed values (V_Curr), the newly calculated partial aggregate value ($NewAggr$) and the last transmitted partial aggregate value of each node (LTA), the difference between these two values ($Diff$), and whether the node makes a transmission or not based on whether the absolute value of this deviation is greater than the maximum permitted error in the node ($|Diff| > E_i$). Notice that whenever a node makes a transmission, then the values of LTA are modified in the next epoch. Moreover, since we are using the model of TAG, each non-leaf node first receives (any) messages from its children nodes and then calculates the new estimate of its partial aggregate.

5.3. Calculating the potential gain of each node

Our algorithm updates the width of the filter installed in each node by considering the potential gain of increasing the error threshold at a sensor

Table 3
Node operation in residual mode

Node	E_i	Epoch 1					Epoch 2				
		V_Curr	$NewAggr$	LTA	$Diff$	Transmit?	V_Curr	$NewAggr$	LTA	$Diff$	Transmit?
4	1	20	20	19	1	NO	21	21	19	2	YES
5	4	50	50	45	5	YES	51	51	50	1	NO
6	2	10	10	7	3	YES	9	9	10	−1	NO
7	1	25	25	24	1	NO	23	23	24	−1	NO
8	3	12	12	16	−4	YES	17	17	12	5	YES
2	0	—	69 (19 + 50)	64	5	YES	–	71 (21 + 50)	69	2	YES
3	2.5	19	65 (10 + 24 + 12 + 19)	67	−2	NO	17	68 (10 + 24 + 17 + 17)	67	1	NO
1	0	30	166 (69 + 67 + 30)	160	6	N/A	28	166 (71 + 67 + 28)	166	0	N/A

node, which is defined as the amount of messages that we expect to save by allocating more resources to the node. This computation of potential gains, as we will show, requires only local knowledge, where each node simply considers statistics from its children nodes in the aggregation tree.

In Fig. 4 we show the expected behavior of a sensor node N_i, varying the width of its filter W_i. The y-axis plots the number of messages sent from this node to its parent in the aggregation tree in a period of *Upd* epochs. Assuming that the measurement on the node is not constant, a zero width filter ($W_i = E_i = 0$) results in one message for each of the *Upd* epochs. By increasing the width of the filter, the number of messages is reduced, up to the point that no messages are required. Of course, in practice this may never happen as the width of the filter required may exceed the global error constraint *E_Global*. Some additional factors that can make a node deviate from the typical behavior of Fig. 4 also exist. As an example, the measurement of the node may not change for some period of time exceeding *Upd*. In such a case, the curve becomes a straight line at $y = 0$ and no messages are sent (unless there are changes on the subtree rooted at the node). In such cases of very stable nodes, we would like to be able to detect this behavior and redistribute the error to other, more volatile nodes. At the other extreme, node N_i may be so volatile that even a filter of considerable width will not be able to suppress any messages. Thus, the curve becomes a straight line at $y = Upd$. Notice that the same may happen because of a highly volatile node N_j that is a descendant of N_i in the aggregation tree.

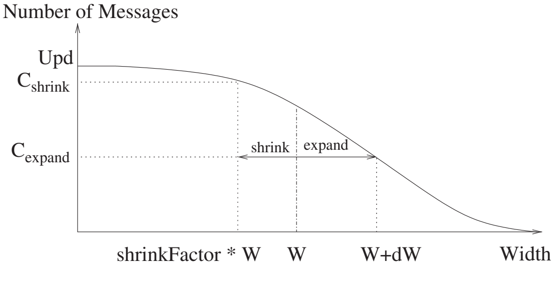

Fig. 4. Potential gain of a node.

In principle, we cannot fully predict the behavior of a node N_i unless we take into account its interaction will all the other nodes in its subtree. Of course, a complete knowledge of this interaction is infeasible, due to the potentially large amounts of information that are required, as described in Section 3.2.1. We will thus achieve this by computing the potential gains of adjusting the width of the node's filter W_i, using simple *local* statistics that we collect during the query evaluation.

Let W_i be the width of the filter installed at node N_i at the last update phase. The node also knows the *shrinkFactor* that is announced when the query is initiated. Unless the adaptive procedure decides to increase the error of the node, its filter's width is scheduled to be reduced to $shrinkFactor \times W_i$ in the next update phase, which takes place every *Upd* epochs. The node can estimate the effects of this change as follows. At the same time that the node uses its filter W_i to decide whether or not to send a message to its parent, it also keeps track of its decision assuming a filter of a smaller width of $shrinkFactor \times W_i$. This requires a single counter C_{shrink} that keeps track of the number of messages that the node would have sent if its filter was reduced. C_{shrink} gives an estimate of the negative effect of reducing the filter of N_i. Since we would also like the node to have a chance to increase its filter, the node also computes the number of messages C_{expand} in case its filter was increased by a factor $\mathrm{d}W$ to be defined later.[3]

Our process is demonstrated in Fig. 4. Let $\delta G \geqslant 0$ be the reduction in the number of messages by changing the width from $shrinkFactor \times W_i$ (which is the default in the next update phase) to $W_i + \mathrm{d}W$. The *potential gain* for the node is defined as

$$Gain_i = \delta G = C_{shrink} - C_{expand}.$$

[3]Even though this computation based on two anchor points may seem simplistic, there is little more that can truly be accomplished with only local knowledge, since the node cannot possibly know exactly which partial aggregates it would have received from its children in the case of either a smaller or a larger filter, because these partial aggregates would themselves depend on the corresponding width changes in the filters of the children nodes.

It is significant to note that our definition of the *potential gain* of a node is independent on whether the node is active or not, since the algorithm for deciding whether to transmit a message or not is only based on the value of the partial aggregate calculated for the node's entire subtree. Moreover, the value of dW is not uniquely defined in our algorithms. In our implementation we are using the following heuristics for the computation of gains:

- For leaf nodes, we use $dW = E\_Global/N_{active}$, N_{active} being the number of active nodes in the aggregation tree.
- For non-leaf nodes, in the residual mode, we need a larger value of dW, since the expansion of the node's filter should be large enough to allow the node to coalesce negative correlations in the changes of the aggregates on its children nodes. As a heuristic, we have been using $dW = num\_children_i \times (E\_Global/N_{active})$, where $num\_children_i$ is the number of children of node N_i.

These values of dW have been shown to work well in practice on a large variety of tested configurations. We need to emphasize here that these values are used to give the algorithm an estimate on the behavior of the sensor and that the actual change in the widths W_i of the filters will also be based on the amount of "error budget" available and the behavior of all the other nodes in the tree.

5.3.1. Computation of cumulative gains

The computation of the potential gains, as explained above, provides us with an idea of the effect that modifying the size of the filter in a node may have, but is by itself inadequate as a metric for the distribution of the available error to the nodes of its subtree. This happens because this metric does not take into account the corresponding gains of descendant nodes in the aggregation tree. Even if a node may have zero potential gain (this may happen, for example, if either the node itself or some of its descendants are very volatile), this does not mean that we cannot reduce the number of transmitted messages in some areas of the subtree rooted at that node.

Because of the top-down redistribution of the errors that our algorithm applies (using the *AdjRoot* algorithm described below), if no budget is allocated to N_i by its parent node then all nodes in the subtree of N_i will not get a chance to increase their error thresholds and this will eventually lead to every node in that subtree to send a new message on each epoch, which is clearly an undesirable situation. Thus, we need a way to compute the *cumulative gain* on the subtree of N_i and base the redistribution process on that value. In our framework we define the cumulative gain on a node N_i as

$$CumGain_i = \begin{cases} Gain_i & N_i : \text{leaf node}, \\ Gain_i + \sum\limits_{N_j \in children(N_i)} CumGain\_Sub[j] & \text{otherwise}. \end{cases}$$

This recursive formula is computed in a bottom-up manner by having nodes piggy-back the value of their cumulative gain in each message that they transmit to their parent along with their partial aggregate value. This is a single number that is being aggregated in a bottom-up manner, and thus poses a minimal overhead. Moreover, transmitting the cumulative gain is necessary only if its value has changed, and in most cases only if this change is significant, since the last transmission of the node.

5.4. Adjusting the filters

We here present two algorithms for adjusting the width of the filters on the nodes. Both algorithms make their decisions using the cumulative gains calculated at each node. They differ in that in the first algorithm, denoted as *AdjRoot*, the *Root* node is the one initiating the process based on the available error budget generated from shrinking the filters. In contrast, in the second algorithm that we denote as *AdjLocal*, this process happens in a localized manner on a level by level basis in the aggregation tree. Below we provide details for both algorithms.

5.4.1. The AdjRoot algorithm

Every *Upd* epochs, all the filters shrink by a factor of *shrinkFactor* (see Algorithm 1, Lines

5–7). This results in an error budget of $E\_Global \times (1 - shrinkFactor)$ which the *Root* node can distribute to the nodes of the tree. Each node N_i can distribute a total error of *E_Additional* to itself and its descendants (Lines 8–10) as follows (for the *Root* node,
$E\_Additional = E\_Global \times (1 - shrinkFactor)$)

- For each subtree j of node N_i, increase E_Sub_j proportionally to its cumulative gain:

$$E\_Additional_j = \frac{E\_Additional \times CumGain\_Sub[j]}{Gain_i + \sum_{N_j \in children(N_i)} CumGain\_Sub[j]}.$$

  This distribution is performed only when this quantity is at least equal to E_Global/N_{active}.
- The remaining error budget is distributed to the node itself.

The fraction of the error budget allocated to the node itself and to each of the subtrees is analogous to the expected benefit of each choice. The use of the computed local gain on the node in comparison to the cumulative gains of its subtrees, allows us to differentiate on the true cause of the transmissions coming out of this node.

The only additional detail is that in case when the error allocated to a subtree of node N_i is less than the E_Global/N_{active} value, then we do not allocate any error in that subtree, and allocate this error to node N_i itself. This is done to avoid sending messages downwards the aggregation tree for adjusting the filters when the error budget is too small.

5.4.2. The AdjLocal algorithm

In the *AdjLocal* algorithm, the nodes negotiate the allocation of the error budget in a localized level-by-level manner, instead of having the whole process initiated by the *Root* node. In particular, each non-leaf node in the tree claims an available error budget equal to

$$E\_Additional_i = \sum_{N_j \in children(N_i)} E_j \times (1 - shrinkFactor). \quad (1)$$

This is exactly the available error budget due to the shrinkage of the filters of its immediate descendants. The allocation of this budget among itself and its children nodes in the tree is performed using the potential gain of the node and the gains of its subtrees:

$$E\_Additional_j = \frac{E\_Additional_i \times Gain_j}{Gain_i + \sum_{N_j \in children(N_i)} Gain_j}. \quad (2)$$

One way to visualize the differences of the two algorithms is to consider how the error budget is being distributed. In the *AdjRoot* algorithm, the whole budget is claimed by the *Root* node. This is possible because all nodes shrink their filters by the same percentage. Then, this error budget is let to flow downwards through the tree, using the accumulated statistics (gains) on the nodes. This process continues until either we reach a leaf node, or when the remaining budget is too small. In the later case the node in consideration claims all the remaining error budget, thus saving downward messages on the corresponding subtree. In contrast, the *AdjLocal* algorithm adjusts the filters in a localized fashion. Any intermediate node in the aggregation tree uses information on the filter widths of its direct descendant nodes to determine its available error budget and then distributes this budget among them and the node itself, without recursively continuing this process on lower levels of the tree.

When comparing the *AdjRoot* and the *AdjLocal* algorithms, one would expect in most cases the *AdjRoot* algorithm to perform better, as it allows broader redistribution of the available error budget. For instance, the *AdjLocal* algorithm will require more rounds (update periods) to shift a significant amount of error from a subtrees S_1 rooted at a node close to the *Root* to a sibling subtree S_2, because the error-budget will first have to gradually *ascend* towards the root node of the S_1 subtree and then slowly be distributed to the nodes in the S_2 subtree. In *AdjLocal*, whenever some nodes allocate a significant amount of their error budget to themselves, then this results in an increased error budget for the parents of these nodes in the next update period. Using this process, the error of an entire subtree can gradually ascend to (and therefore be distributed by) nodes in higher levels of the aggregation tree.

However, there are occasions when we expect the *AdjLocal* algorithm to be superior. In particular, consider the case when the *Root* node is physically located very far from the nodes that actually collect measurements and that the aggregation tree is tall and narrow in its upper levels. This is a realistic scenario when the aggregate query involves the values observed in just one area of the network. In some extreme cases, the *Root* will be connected to the active nodes through a string of nodes. When the *Root* is several links away from the leaf nodes, the *AdjRoot* algorithm requires a lot of messages to propagate the error budget to the nodes that actually need it. In such cases, the *AdjLocal* algorithm might require fewer messages, since the redistribution process will mostly involve active nodes at (or near) the leaves of the tree. Moreover, due to the minimum additional error that can be distributed to subtrees by the *AdjRoot* algorithm, nodes with modest gains may not receive any budget if they belong to subtrees with small cumulative gains. However, in the *AdjLocal* algorithm, through a local redistribution of errors from their siblings and their parent, these nodes will still be able to increase their filters and, thus, reduce the number of their transmitted messages.

6. Experiments

We have developed a simulator for sensor networks that allows us to vary several parameters like the number and configuration of the nodes, the topology of the aggregation tree, the data distribution etc. The synchronization of the sensor nodes is performed as described in TAG [2]. In our experiments we compare the following algorithms:

1. *BBA*: This is an implementation of the algorithm presented in [8] for the adaptive precision setting of cached approximate values.
2. *Uni*: This is a static setting where the error is evenly distributed among all active sensor nodes and, therefore, does not incur any communication overhead for adjusting the error thresholds of the nodes.
3. *PGA* (Potential Gains Adjustment): This is our precision control algorithm, based on the potential gains as described in Section 5. For adjusting the filters of the sensor nodes we use the *AdjRoot* algorithm; later in this section we also provide an experimental evaluation with the *AdjLocal* method as well.

For the *BBA* algorithm, we experimented with several heuristics for estimating the cost C_i of each message transmitted by a node N_i, and set it in our experiments to $(dist_i + 1)/2$, where $dist_i$ denotes the distance in number of hops of the node from the *Root* node. Our heuristic is the average of the worst case cost (message not aggregated with any other message until it reaches the *Root*) and the best case cost (message aggregated with others at the parent node of N_i) of messages transmitted by node N_i, and provided the best results in most cases. With this heuristic, each node is able to estimate its burden score and potentially transmit it to the *Root* node at the last epoch of the update period. It is important to emphasize that in our implementation of *BBA*, we do not account for the additional amount of information needed for the nodes to transmit their burden scores (we do not count the messages needed to transmit them). This is an ideal scenario for *BBA* and is used to provide a more direct comparison to the *PGA* algorithm, as to the amount of messages pruned by each method due to the installation of the filters.

For the *PGA* and *BBA* algorithms we made a few preliminary runs to choose their internal parameters *adjustment period* (*Upd*) and *shrink percentage* (*shrinkFactor*). Based on the observed behavior of the algorithms, we have selected the combination of values of Table 4 as the most representative ones for revealing the "preferences" of each algorithm. The first configuration (Conf1) consistently produced good results, in a variety of tree topologies and data sets, for the *PGA* algorithm, while the second configuration (Conf2) was typically the best choice for the *BBA* algorithm. In the *BBA* algorithm we also determined experimentally that distributing the

Table 4
Used configurations

Parameters	Configuration	
	Conf1	Conf2
Upd	50	20
shrinkFactor	0.6	0.95
Invocations	Fewer	Frequent
Error amount redistributed	Significant	Smaller

available error to 10% of the nodes with the highest burden scores was the best choice for the algorithm.

The initial allocation of error thresholds was done using the uniform policy. We then used the first 10% of epochs as a warm-up period for the algorithms to adjust their thresholds and report the results (number of transmitted messages) for the later 90%.

6.1. Description of data sets

6.1.1. Synthetic data sets

We generated synthetic data, similar in spirit to the data used in [8]. For each simulated active node, we generated values following a random walk pattern, each with a randomly assigned step size in the range $(0 \ldots 2]$. We further added in the mix a set of "unstable nodes" whose step size is much larger: $(0 \ldots 200]$. These volatile nodes allow us to investigate how the different algorithms adapt to noisy sensors. Ideally, when the step-size of a node is comparable to the global error threshold, we would like the precision control algorithm to restrain from giving any of the available budget to that node at the expense of all the other sensor nodes in the tree. We denote with $P_{unstable}$ the probability of an active node being unstable.

We further divide the sensor nodes in two additional classes: *workaholics* and *regulars*. Regular sensors make a random step with a fixed probability of 1% during an epoch. Workaholics, on the other hand, make a random step on every epoch. We denote with $P_{workaholic}$ the probability of an active node being workaholic.

6.1.2. Real data sets

We also report results using two real data sets. The first, denoted as `LBL-TCP-3`, is described in [28] and was also used in the original paper of [8]. It contains information on all the wide-area TCP traffic between the Lawrence Berkeley Laboratory and the rest of the world for a period of 2 h. We have processed this data and created individual time-series (one per sensor node) for each of the 1540 source IP addresses in the trace. Each time-series describes the number of bytes transmitted from a source IP per second.

The second real data set, denoted as `Weather`, was obtained from IRI/LDEO Climate Data Library and consists of precipitation data from 1582 weather stations. Again, we created individual time-series (one per sensor node) using precipitation measurements from each weather station. Sensor networks that are used in environmental monitoring are expected to process similar data.

6.2. Network topology

We used three different network topologies denoted as T_{leaves}, T_{all} and T_{random}. In T_{leaves} the aggregation tree was a balanced tree with 5 levels and a fan-out of 4 (341 nodes overall). For this configuration all active nodes were at the leaves of the tree. In T_{all}, for the same tree topology, all nodes (including the *Root*) were active. Finally in T_{random} we used 500 sensor nodes, forming a random tree each time. The maximum fan-out of a node was in that case 8 and the maximum depth of the tree 6. Intermediate nodes in T_{random} were active with probability 20%.

In all experiments, we executed the simulator 10 times and present here the averages. In all runs we used the *SUM* aggregate function (the performance of *AVG* was similar).

6.3. Benefits of residual mode of operation

The three precision control algorithms considered (*Uni*, *PGA*, *BBA*) along with the mode of operation (residual: *Res*, non-residual: *NoRes*) provide us with six different choices ($Uni+Res$, $Uni+NoRes$, ...). We note that $BBA+NoRes$ is the original algorithm of [8] running over TAG, while $BBA + Res$ is our extension of that algorithm using the residual mode of operation. The combination $PGA + Res$ denotes our algorithm. In this first experiment we investigate whether the precision control algorithms benefit from the use of the residual mode of operation. We also seek their preferences in terms of the values of parameters *adjustment period* and *shrink percentage*.

We used a synthetic data set with $P_{unstable} = 0$ and $P_{workaholic} = 0.2$. We then let the sensors operate for 40,000 epochs using a fixed error constraint $E\_Global = 500$. The average value of the SUM aggregate was 25,600, meaning that this E_Global value corresponds to a relative error of about 2%. In Table 5 we show the total number of messages in the sensor network for each choice of algorithm and tree topology and each selection of parameters. We also show the number of messages for an exact computation of the SUM aggregate using one more method, entitled as $(E\_Global = 0) + Res$, which places a zero width filter in every node and uses our residual mode of operation for propagating changes. Effectively, a node sends a message to its parent only when the partial aggregate on its subtree changes. This is nothing more than a slightly enhanced version of TAG. The following observations are made:

- Using a modest E_Global value of 500 (2% relative error), we are able to reduce the number of messages by 7.6–9.9 times (in $PGA + Res$) compared to $(E\_Global = 0) + Res$. Thus, error-tolerant applications can significantly reduce the number of messages in the network resulting in great savings on both bandwidth and energy consumption.
- Algorithm PGA seems to require fewer invocations (larger *adjustment period*) but with a larger percentage of the error to be redistributed (a smaller *shrink percentage* results in a wider reorganization of the error thresholds). In the table we see that the number of messages for the selection of values of Conf1 is always smaller. Intuitively, larger adjustment periods allow for more reliable statistics on the computation of potential gains.
- On the contrary, BBA seems to behave better when filters are adjusted more often by small increments. We also note that BBA results in a lot more messages than PGA, no matter which configuration is used.
- The PGA algorithm, when using the residual operation ($PGA + Res$), results in substantially fewer messages than all the other alternatives. Even when using the non-residual mode of operation, PGA outperforms, significantly, the competitive algorithms.
- BBA seems to benefit only occasionally from the use of the residual operation. The adjustment of thresholds based on the burden of a node cannot distinguish on the true cause of a transmission (change on local measurement or change in the subtree) and does not seem to provide a good method of adjusting the filters with respect to the tree hierarchy.

Table 5
First number is total number of messages (in thousands) in the network when using parameters of Conf1, second for Conf2 (see also Table 4).

	T_{leaves}	T_{all}	T_{random}
$PGA + Res$	**423**/978	**479**/903	**677**/1207
$PGA + NoRes$	463/924	558/894	830/1454
$BBA + Res$	2744/1654	2471/**1426**	3775/2657
$BBA + NoRes$	3203/**1394**	2967/1481	4229/**2474**
$Uni + Res$	**2568**	**2451**	**3906**
$Uni + NoRes$	**2568**	2642	4044
$(E\_Global = 0) + Res$	**4176**	**4176**	**5142**

Uni does not use these parameters. Best numbers for each algorithm in bold.

In the rest of the section, for PGA we used the residual mode of operation. For BBA we tested both the residual and non-residual modes and present the best results for each experiment. We configured PGA using the values of Conf1 and BBA using the values of Conf2.

6.4. Sensitivity analysis

We first investigate the performance of the algorithms when varying $P_{workaholic}$ and $P_{unstable}$. We first fixed $P_{workaholic}$ to be 20% and used

$P_{unstable} = 0$. In Fig. 5 we plot the total number of messages in the network (y-axis) for 40,000 epochs when varying the error constraint E_Global from 100 to 2000 (8% is terms of relative error). Depending on E_Global, PGA results in up to 4.8 times fewer messages than BBA and up to 6.4 times fewer than Uni. Figs. 6 and 7 repeat the experiment for the T_{all} and T_{random} configurations.

In Fig. 8 we vary $P_{workaholic}$ between 0 and 1 for T_{all} (best network topology for BBA) and for $E\_Global = 500$. Again PGA outperforms the other algorithms. An important observation is that when the value of $P_{workaholic}$ is either 0 or 1, all the methods behave similarly. In this case all the nodes in the network have the same characteristics, so it is not surprising that Uni performs so well. The PGA and BBA algorithms managed to filter just a few more messages than Uni for these

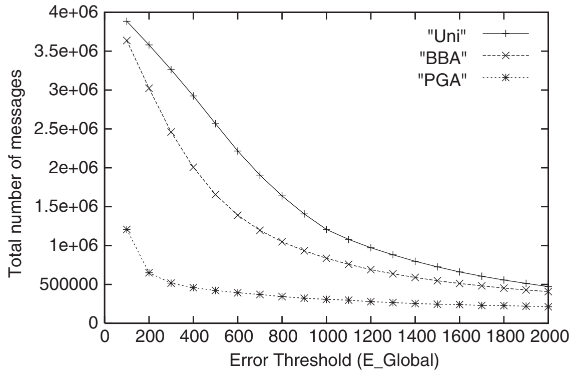

Fig. 5. Messages varying E_Global, T_{leaves} configuration.

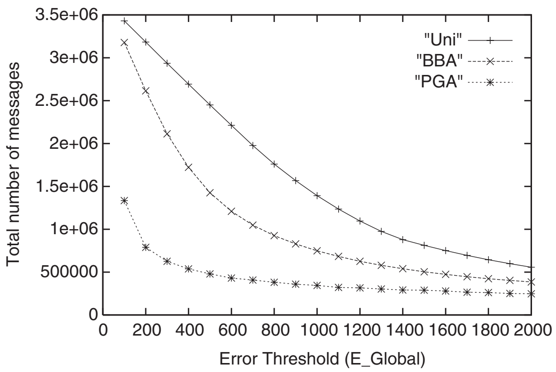

Fig. 6. Messages varying E_Global, T_{all} configuration.

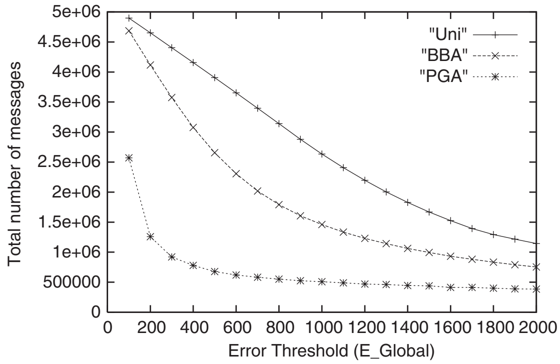

Fig. 7. Messages varying E_Global, T_{random} configuration.

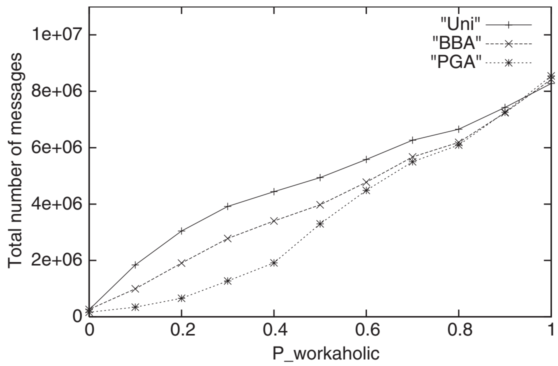

Fig. 8. Messages varying $P_{workaholic}$, T_{all} configuration.

cases, but due to their overhead for updating the error thresholds of the nodes, the overall number of transmitted messages was about the same for all techniques.

In Figs. 9 and 10 we vary the percentage of unstable nodes (nodes that make very large steps) from 0% to 100% and plot the total number of messages for T_{all} and T_{random} ($P_{workaholic} = 0.2$, $E\_Global = 500$). For $P_{unstable} = 1$ the error threshold (500) is too small to have an effect on the number of messages and all algorithms have practically the same behavior. For smaller values of $P_{unstable}$, algorithm PGA results in a reduction in the total number of messages by a factor of up to 3.8 and 5.5 compared to BBA and Uni, respectively.

In Fig. 11 we vary the value of Upd from 10 to 100 for a running query of 1000 epochs ($P_{workaholic} = 20\%$, $E\_Global = 500$). For both

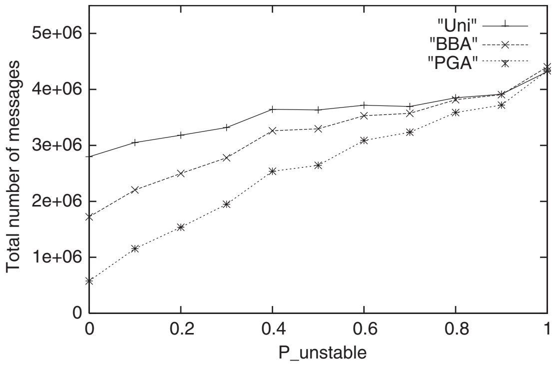

Fig. 9. Messages varying $P_{unstable}$, T_{all} configuration.

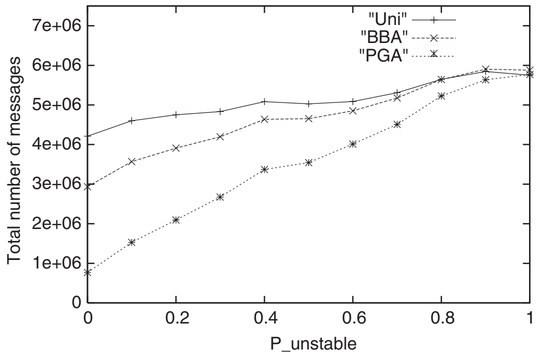

Fig. 10. Messages varying $P_{unstable}$, T_{random} configuration.

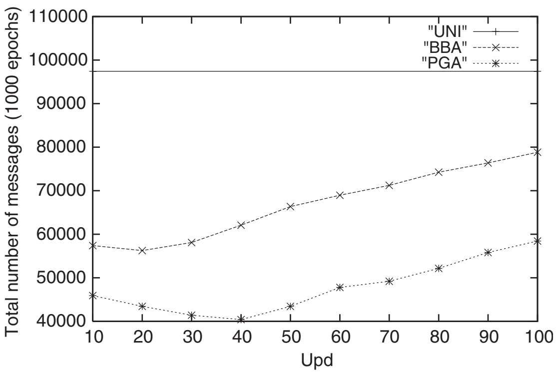

Fig. 11. Messages varying *Upd*.

PGA and *BBA* the number of messages is initially reduced with increasing values of *Upd*. However, in both algorithms there is a point where a further increase in the value of *Upd* results in more messages since there are not enough update phases to properly adjust the behavior of the nodes. We

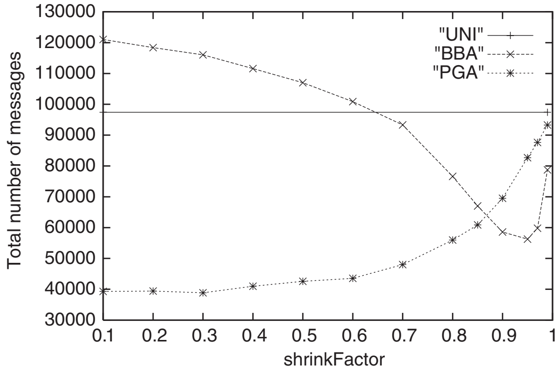

Fig. 12. Messages varying *shrinkFactor*.

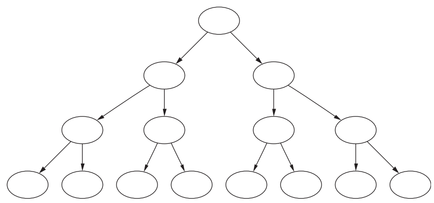

Fig. 13. Original tree.

further varied *shrinkFactor* between 10% and 99%. The results in Fig. 12 suggest that *BBA* behaves better when frequent updates (Fig. 11) reallocate small amount of the error budget. Algorithm *PGA* shows a relatively steady behavior for *shrinkFactor* between 10% and 70%.

We further examine how all algorithms scale with the size of the aggregation tree and in particular with the distance of the leave nodes from the *Root* (changing the fan-out of the aggregation tree did not significantly affect performance). We started with a balanced aggregation tree with a fan-out of 4 and 6 levels (1365 nodes overall) having all active nodes at the leaves of the tree (i.e. similar to T_{leaves}). We then gradually augmented the tree by injecting *transport* nodes between levels 0–1, 1–2 and 2–3 in the tree. This process is illustrated in Figs. 13–15. For presentation purposes in these figures we use an initial tree with fan-out 2 and only 4 levels. In Fig. 14 we show the resulting tree of adding transport nodes between levels 0–1 and 1–2, while Fig. 15 shows the tree after adding another set of transport nodes between these levels. Essentially each step makes the top-level nodes of the tree lay further away for the leaf nodes that collect the measurements.

In Fig. 16 we compare the performance of the *BBA* algorithm against *PGA* (residual mode) with the later using (i) the *AdjRoot* algorithm for adjusting the filters (the default choice) and (ii) the *AdjLocal* algorithm. The y-axis is the figure shows the percentage of nodes in the tree transmitting on an epoch, averaged over 1000 epochs ($E\_Global = 1500$). The x-axis shows the number of successive steps of adding transport nodes. As more nodes are added between the *Root* and the leaves of the tree, the number of messages increases in both *BBA* and $PGA + AdjRoot$ algorithms. This is due to both

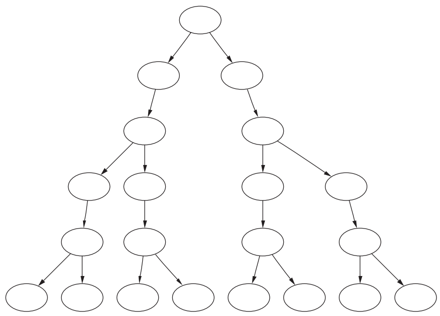

Fig. 14. After adding transport nodes between layers 0–1 and 1–2.

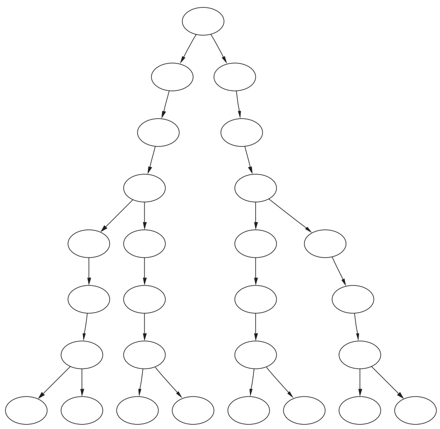

Fig. 15. After adding second set of transport nodes between layers 0–1 and 1–2.

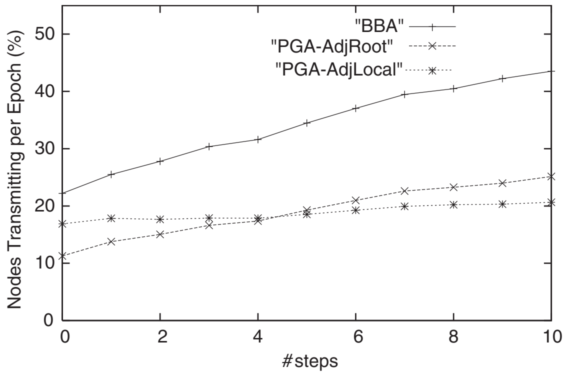

Fig. 16. Messages varying the number of transport layers.

the increased number of nodes in the tree and because both algorithms adjust the filters in a top-down manner, thus resulting in a larger re-organization overhead, since the average distance of the *Root* node from the nodes of the aggregation tree that ultimately received most of the error budget increases. In contrast, when using the *AdjLocal* algorithm for adjusting the filters, the performance is practically unaffected by the addition of the transport nodes. We note that both $PGA + AdjLocal$ and $PGA + AdjRoot$ operate on the same set of statistics collected at the nodes and, in principle, one can alternate between the two algorithms, i.e. use $PGA + AdjLocal$ when the data distribution appears to be relatively static and switch to $PGA + AdjRoot$ when a quick large-scale redistribution of the budget is required.

6.5. Experiments with real data

In Fig. 17 we summarize our results for the `LBL-TCP-3` data set and the T_{random} topology. This data set has the unique feature that many IP-sources show long periods of inactivity (number of bytes sent is zero) followed by short, bursty transmissions. We include this data, as a "hard" case for our algorithm, since this property makes it hard to predict future behavior based on past statistics. However, we can see that *PGA* still outperforms the other alternatives. We also note that, for very small values of *E_Global*, *Uni* is very competitive, as in that case the available thresholds are not enough to prune transmissions of

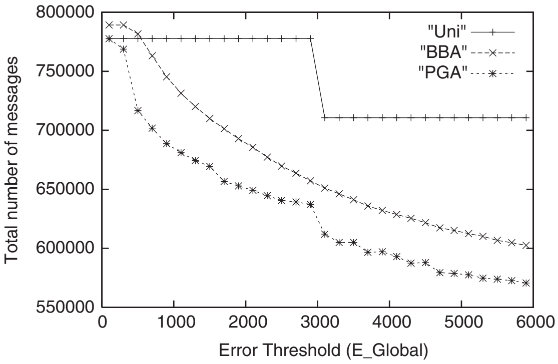

Fig. 17. Messages, LBL-TCP-3 data set.

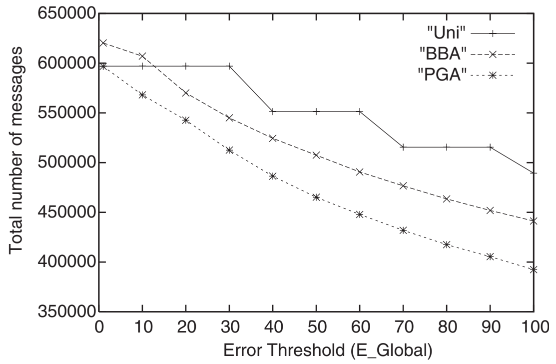

Fig. 18. Messages, Weather data set.

active IP sources. We remind that *Uni* has a static allocation of filters, and has no overhead of adjusting them, unlike the other two algorithms. We also provide results using precipitation readings from the Weather data set. In Fig. 18 we show the total number of messages, varying *E_Global*, for the three algorithms, when nodes are organized in the T_{all} configuration.

7. Conclusions and future directions

In this paper, we proposed a new framework for approximate in-network data aggregation for sensor networks. Unlike previous approaches, our algorithms exploit the tree hierarchy of the sensor nodes to significantly reduce the number of transmitted messages, and therefore, increase the lifetime of the network.

Our algorithms are based on two key ideas that we presented in this paper. Firstly, the residual mode of operation for nodes in the aggregation tree allows nodes to apply their error filters to the partial aggregates of their subtrees, and therefore, potentially suppress messages from being transmitted towards the root node of the tree. A second key idea is the use of simple and local statistics to estimate the potential gain of allocating additional error to nodes in a subtree. This is a significant improvement over straightforward extensions for the hierarchical setting of previous approaches that require a large amount of information to be transmitted to the root node of the tree. Through an extensive set of experiments, we have shown in this paper that while the distribution of the error based on the computed gains is the major factor for the effectiveness of our techniques compared to other approaches, the fusion of the two ideas provides the best improvements.

While our algorithms have been shown to drastically reduce the number of messages exchanged among the nodes, there is still a number of open issues to explore. Since multiple monitoring nodes may exist in sensor networks, we plan to consider how to extend our techniques to optimize the bandwidth utilization of multiple continuous queries. Moreover, in many applications the dual problem might be of interest, that is to minimize the error of the approximation for a target bandwidth constraint. In other cases, minimizing a weighted sum of error and bandwidth might be desirable. Finally, for applications that require more powerful sensor nodes, it will be challenging to devise more complex error filters (perhaps by building appropriate data models [16]) than the ones used in this manuscript.

References

[1] A. Ailamaki, C. Faloutsos, P.S. Fischbeck, M.J. Small, J. VanBriesen, An environmental sensor network to determine drinking water quality and security, ACM SIGMOD Rec. 32 (4) (2003) 47–52.

[2] S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, Tag: a tiny aggregation service for ad hoc sensor networks, in: OSDI Conference, 2002.

[3] S. Madden, M.J. Franklin, J.M. Hellerstein, W. Hong, The design of an acquisitional query processor for sensor networks, in: ACM SIGMOD Conference, June 2003.
[4] D. Estrin, R. Govindan, J. Heidermann, S. Kumar, Next century challenges: Scalable coordination in sensor networks, in: MobiCOM, 1999.
[5] C. Intanagonwiwat, D. Estrin, R. Govindan, J. Heidermann, Impact of network density on data aggregation in wireless sensor networks, in: ICDCS, 2002.
[6] M.A. Sharaf, J. Beaver, A. Labrinidis, P.K. Chrysanthis, TiNA: a scheme for temporal coherency-aware in-network aggregation, in: Proceedings of ACM MobiDE Workshop, September 2003.
[7] Y. Yao, J. Gehrke, The Cougar approach to in-network query processing in sensor networks, SIGMOD Rec. 31 (3) (2002) 9–18.
[8] C. Olston, J. Jiang, J. Widom, Adaptive filters for continuous queries over distributed data streams, in: ACM SIGMOD Conference, 2003, pp. 563–574.
[9] A. Cerpa, D. Estrin, ASCENT: adaptive self-configuring sensor network topologies, in: INFOCOM, 2002.
[10] J. Heidermann, F. Silva, C. Intanagonwiwat, R. Govindanand, D. Estrin, D. Ganesan, Building efficient wireless sensor networks with low-level naming, in: SOSP, 2001.
[11] A. Demers, J. Gehrke, R. Rajaraman, N. Trigoni, Y. Yao, The Cougar project: A WorkIn Progress report, ACM SIGMOD Rec. 32 (4) (2003) 53–59.
[12] A. Ghose, J. Grossklags, J. Chuang, Resilient data-centric storage in wireless ad-hoc sensor networks, in: Mobile Data Management, 2003, pp. 45–62.
[13] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, S. Shenker, GHT: a geographic hash table for data-centric storage, in: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications, 2002, pp. 78–87.
[14] J.-H. Chang, L. Tassiulas, Energy conserving routing in wireless ad-hoc networks, in: INFOCOM, 2000, pp. 22–31.
[15] S. Singh, M. Woo, C.S. Raghavendra, Power-aware routing in mobile ad hoc networks, in: Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1998, pp. 181–190.
[16] Y. Kotidis, Snapshot queries: towards data-centric sensor networks, in: Proceedings of ICDE, 2005.
[17] J. Considine, F. Li, G. Kollios, J. Byers, Approximate aggregation techniques for sensor databases, in: Proceedings of ICDE, 2004.
[18] J. Chen, D.J. Dewitt, F. Tian, Y. Wang, Niagara CQ: a scalable continuous query system for internet databases, in: ACM SIGMOD Conference, 2000.
[19] J.M. Hellerstein, M.J. Franklin, S. Chandrasekaran, A. Descpande, K. Hildrum, S. Madden, V. Raman, M.A. Shah, Adaptive query processing: technology in evolution, IEEE Data Eng. Bull. 23 (2) (2000).
[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma, Query processing, resource management, and approximation in a data stream management system, in: CIDR, January 2003, pp. 245–256.
[21] C. Olston, B.T. Loo, J. Widom, Adaptive precision setting for cached approximate value, in: ACM SIGMOD Conference, 2001.
[22] C. Olston, J. Widom, Offering a precision-performance tradeoff for aggregation queries over replicated data, in: VLDB Conference, 2000, pp. 144–155.
[23] C. Olston, J. Widom, Best-effort cache synchronization with source cooperation, in: ACM SIGMOD Conference, 2002, pp. 73–84.
[24] D. Barbará, H. Garcia-Molina, The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems, in: Proceedings of EDBT, 1992, pp. 23–27.
[25] N. Soparkar, A. Silberschatz, Data-value partitioning and virtual messages, in: Proceedings of PODS, Nashville, Tennessee, April 1990, pp. 357–367.
[26] R. Cheng, D.V. Kalashnikov, S. Prabhakar, Evaluating probabilistic queries over imprecise data, in: ACM SIGMOD Conference, 2003, pp. 551–562.
[27] A. Deligiannakis, Y. Kotidis, N. Roussopoulos, Compressing historical information in sensor networks, in: Proceedings of ACM SIGMOD Conference, 2004.
[28] V. Paxson, S. Floyd, Wide-area traffic: the failure of Poisson modeling, IEEE/ACM Trans. Networking 3 (3) (1995) 226–244.