Asynchronous In-network Prediction: Efficient Aggregation in Sensor Networks

PAVAN EDARA, ASHWIN LIMAYE and KRITHI RAMAMRITHAM
Indian Institute of Technology Bombay
{pavan,ashwin,krithi}@cse.iitb.ac.in

Given a sensor network and aggregate queries over the values sensed by subsets of nodes in the network, how do we ensure that *high quality results* are served for the *maximum possible time*? The issues underlying this question relate to the *fidelity* of query results and *lifetime* of the network. To maximize both, we propose a novel technique called *asynchronous in-network prediction* incorporating two computationally efficient methods for in-network prediction of partial aggregate values. These values are propagated via a tree whose construction is cognizant of (a) the coherency requirements associated with the queries, (b) the remaining energy at the sensors, and (c) the communication and message processing delays. Finally, we exploit *in-network filtering* and *in-network aggregation* to reduce the energy consumption of the nodes in the network. Experimental results over real world data support our claim that for aggregate queries with associated coherency requirements, a prediction based asynchronous scheme provides higher quality results for a longer amount of time than a synchronous scheme. Also, whereas aggregate dissemination techniques proposed so far for sensor networks appear to have to trade-off quality of data for energy efficiency, we demonstrate that this is not always necessary.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Systems–Sensor Databases

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: aggregation, coherency, energy efficient, prediction, query processing

1. INTRODUCTION

Networks made up of sensor nodes are being increasingly deployed for observing continuously changing data. Sensing devices being battery powered, *energy efficiency* is a primary design consideration: since energy expended for communication is significantly higher than that for local computations, data dissemination techniques should minimize the amount of communication so as to increase the lifetime of the network. Consider executing an aggregate query over values sensed by sensor nodes in a certain region of the network, referred to as the *target region*. The nodes in the target region are called *sources* and the node at which the query is injected into the network is called a *query node*. A query requests the value of an aggregate with an associated *coherency*, c. This denotes the accuracy of the results delivered to the query node relative to that at the sources, and thus, constitutes the user-specified requirement. For example, a query injected for building monitoring is: "Report the average temperature of the southern wall of the building whenever it changes by more than 2^oC". Thus, any change in the average temperature value that is within two degrees of what the query node knows need not be reported and the current value known to the query node is considered accurate enough. User specified coherency requirements are exploited by our approach to extend the lifetime of sensor networks. In scenarios such as this, we note that it is required that changes to the values of data be made available for online decision making.

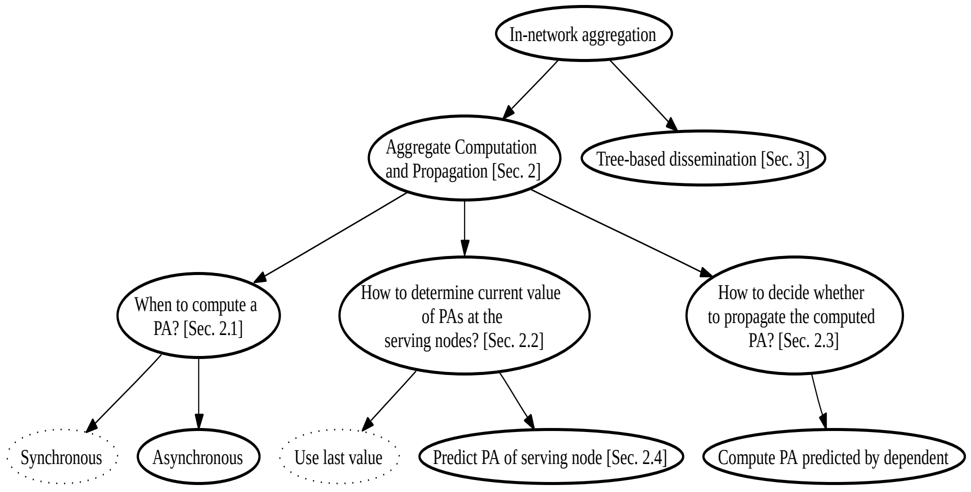

In-network aggregation
Aggregate Computation and Propagation [Sec. 2]
Tree-based dissemination [Sec. 3]
When to compute a PA? [Sec. 2.1]
How to determine current value of PAs at the serving nodes? [Sec. 2.2]
How to decide whether to propagate the computed PA? [Sec. 2.3]
Synchronous
Asynchronous
Use last value
Predict PA of serving node [Sec. 2.4]
Compute PA predicted by dependent

Fig. 1. Summary of issues in handling aggregate queries in sensor networks. (A node in the aggregation tree computes a partial aggregate, PA, based on partial aggregates sent by its serving nodes and forwards it to its dependent)

Contributions of this paper: Given a network of sensors each equipped with a certain amount of initial energy, and an aggregate query with a certain coherency requirement, how should the sensed data be routed from data sources in the query's target region to the query node such that two metrics are maximized:

— *Lifetime*: Assuming that every sensor in the network has a fixed amount of energy which can not be replenished once exhausted, a sensor will stop working after a certain period. This period is defined as Sensor Lifetime. The *query lifetime* is a function[1] of the duration, after the query is injected into the network, for which the query node gets continuous updates from the sensor network.

— *Fidelity*: The quality of delivered query results, i.e., Fidelity is quantitatively measured as the fraction of the query lifetime for which the reported query results are temporally coherent [Shah et al. 2003]. To understand temporal coherency,

[1]In our experiments this is a function of the fraction of the original number of source nodes that contribute to the query result i.e. it accounts for the effect of failure of source nodes on the query result

consider a scenario where the user obtains data items from sensor nodes. The system must track dynamically changing data so as to provide users with temporally coherent information. Now, if the user specifies a temporal coherence requirement (c) for the data item of interest, c denotes the maximum allowable deviation from the value at the source, and thus denotes user-specified tolerance. c is specified in terms of the value of the data item. To maintain coherence, the value of the data seen by the user must be refreshed in such a way that the user-specified coherency requirements are maintained. If $S(t)$ denotes the value of the data item at the source nodes and $U(t)$, the value of the data item known to the user at time t, the system is said to be temporally coherent at time t if $|U(t) - S(t)| \leq c$.

Reducing the number of messages improves both fidelity (by reducing the probability of message losses due to collisions) and lifetime (by reducing energy needs). For many types of aggregate queries, the number of message transmissions can be reduced significantly by computing *partial aggregates* wherever possible while the messages are being routed towards the query node. This technique called *in-network aggregation* has been exploited by [Madden et al. 2002; Yao and Gehrke 2002] to increase lifetime of aggregate queries. The nodes at which this is done are called *aggregator nodes.* Figure 1 is a pictorial representation of issues to be handled in answering an aggregate query using in-network aggregation. Existing approaches to answering coherency based aggregate queries perform in-network aggregation by synchronizing transmissions of nodes level-by-level on an aggregation tree [Sharaf et al. 2003]. Any message that is received by an aggregator node is delayed for a certain amount of time before it can be propagated up the tree. This leads to a definite loss in fidelity. Moreover, these approaches do not address the issues of energy efficiency and timeliness of query results in their tree construction mechanisms. To summarize, our contributions are:

(1) We present AP – an Asynchronous Prediction-based approach for answering aggregate queries. AP incorporates the following novel ingredients:
(a) It makes use of asynchronous in-network aggregation wherein an aggregator node computes a partial aggregate *asynchronously*, i.e., whenever an update that may affect the current partial aggregate is received from one of its serving nodes in the aggregation tree. Existing approaches compute aggregates *synchronously*, often delaying propagation of the effect of received partial aggregates.
(b) When an aggregator node receives a partial aggregate from a serving node and computes a new partial aggregate, what values should it assume for all other serving nodes? In AP the aggregator node predicts these values from the previously received values using a computationally efficient prediction mechanism. This *prediction-based* approach is in contrast to existing *last-value-based* approaches that use the last received values for this purpose.
(c) If each partial aggregate computed as above were to be disseminated towards the source, would it not lead to significant energy consumption? This is where the idea of *in-network prediction* proves to be useful again. When an aggregator node computes a partial aggregate asynchronously, it also calculates the value of the partial aggregate as would be predicted by the receiving node. If the difference between the two values is within a fraction of the coherency

associated with the partial aggregate (derived from user specified coherency on the aggregate), it does not send the computed value, thus saving energy in transmissions.

(2) We propose a tree construction algorithm that has the following features:
(a) It takes into account the coherency requirements associated with the query, the remaining energy at the sensors, and the communication and the message processing delays, thereby contributing to higher lifetime and fidelity of query results.
(b) It is able to exploit the presence of common sources across multiple queries. This leads to further increase in fidelity and lifetime.
(c) It incorporates optimizations to efficiently handle complex aggregate queries with `group by` clauses.
(d) Upon the death of a node, the dissemination tree can be locally adjusted allowing query results to be provided. This increases lifetime.

	Low Fidelity	High Fidelity
Low Lifetime		AL
High Lifetime	SL, SP	AP

Table I. Classification of aggregation schemes

Existing approaches, e.g., [Sharaf et al. 2003], can be classified as preferring synchronous last-value-based (SL) aggregation. Experimental results demonstrate that AP has only one fifteenth of the fidelity loss along with a 40% improvement in lifetime compared to a synchronous last-value-based aggregate computation method. Between the SL approach and our AP approach lie synchronous prediction-based (SP) and asynchronous last-value-based (AL) schemes. Table I tabulates how these schemes perform: we can see that the two building blocks of AP, asynchronous computation and prediction-based in-network aggregation, form a winning combination.

Roadmap. Section 2 presents our approach for computing aggregates asynchronously using in-network prediction. We present our aggregation tree construction algorithm in Section 3. Experimental results are discussed in Section 4. Section 5 surveys related work. Section 6 offers a summary and directions for future work.

2. ASYNCHRONOUS PREDICTION-BASED AGGREGATE COMPUTATION: AN OVERVIEW

Figure 2 shows an example aggregation tree. How such a tree is constructed, given an aggregate query over a target region within a sensor network, is the subject of Section 3. Q is the query node; B, D, E, F are aggregator nodes. H, I, J, K, L, M, N, O, P are the source nodes. In addition, nodes A, B, C, D, E, F, G are referred to as intermediate nodes. The direction of the arrows shows the direction of flow of data from sources to the query node. Note that in general some of the intermediate nodes could themselves be sources. Given such an aggregation tree, the aggregate is computed as follows: Every aggregator node in the tree computes a partial aggregate from the values received from its *serving nodes*. For e.g., node

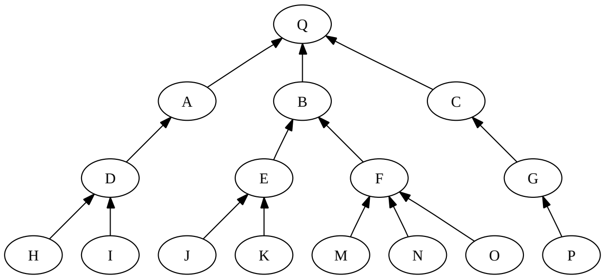


Fig. 2. An example aggregation tree

B computes the partial aggregate of values at serving nodes E and F. Similarly E computes the partial aggregate of values at nodes J and K whereas F computes the partial aggregate of values at M, N and O. Thus, each node in the tree computes a partial aggregate of the values sensed by the source nodes which belong to the subtree rooted at that node. The computed value of the partial aggregate is pushed by a node to its parent node in the aggregation tree, called its *dependent*. For e.g., Q is the dependent of B. In addition to in-network aggregation AP also uses *in-network filtering* whereby a data or partial aggregate value is disseminated to another node only if it differs from the last sent value by more than the associated coherency. We denote by c, the user specified coherency on the aggregate query and by c', the coherency associated with the partial aggregate. c and c' are related as:

$$c' = c \times \beta$$

where β is a real number in the interval $[0, 1]$. To understand β, consider an aggregation tree. In an aggregation tree, the number of messages exchanged between nodes increases as we traverse from the leaf nodes in the tree towards the root. This can be explained by the reduction in the number of messages due to in-network aggregation. Due to higher number of messages near the source nodes, the possibility of messages being lost due to collisions is higher. We use coherency c to suppress the number of messages exchanged between nodes. β effectively controls the number of messages suppressed and ensured that the query node receives results at the desired accuracy.

The sensing activity of the nodes in a sensor network need not be synchronized. As a result, the partial aggregates from different serving nodes may be received at different instants. Whenever an aggregator node receives a partial aggregate (from a serving node), it needs to make the following decisions:

—When to compute its own partial aggregate? Section 2.1 explains our solution to this problem.

—How to determine the current values of partial aggregates at the serving nodes? We discuss our approach in Section 2.2.

—How to decide whether or not to propagate the computed partial aggregate to its dependent? We address this problem in Section 2.3.

Figure 3 shows the pseudocode for our algorithm.

2.1 Asynchronous computation of partial aggregates

In order to provide 100% fidelity, every received message that is required to be sent to the query node should be pushed up the tree as soon as possible i.e., change in values sensed at the sources must be propagated to the query node as soon as possible. To achieve this, we take an asynchronous approach and compute the value of the partial aggregate whenever a node receives a partial aggregate from one of its serving nodes. According to this approach, in Figure 2, suppose node B receives a partial aggregate from node E. In order to compute a new partial aggregate, B needs the current value of the partial aggregate computed by node F. Since this value is unknown to B, what value should B use for the current value at node F? The next subsection addresses this issue.

2.2 Using prediction of partial aggregates at serving nodes

What values should an aggregator node use for the partial aggregates at its serving nodes for computing the new value of its own partial aggregate? We use the *predicted value* of the partial aggregate at each of these serving nodes based upon past knowledge of the partial aggregates received from them. We defer a discussion of our proposed methods for prediction till Section 2.4. Note that our usage of the term epoch is consistent with [Madden et al. 2002; Sharaf et al. 2003] except that within an epoch partial aggregate transmission occurs asynchronously.

Algorithm 2.1: AP(c, R)

```
Purpose: Report aggregate of values sensed in target region $R$ with coherency $c$
Construct aggregation tree $\mathcal{T}$ for nodes in target region $R$.
for each node $n \in \mathcal{T}$
  do
    When a new value, $v_{new}$ is received from some serving node:
    Predict $p_s$, the value of partial aggregate computed by each remaining serving node $s$ of $n$.
    Compute $v_n$, the value of partial aggregate from $n$ using $v_{new}$ and $p_s$'s.
    Compute $v_p$, the value of the partial aggregate from $n$ as estimated by dependent of $n$ in $\mathcal{T}$
    Push $v_n$ to dependent and set $lastPushTime = CurrentTime$ if:
      $|v_n - v_{last}| > c'$ and
      $|v_p - v_n| > \alpha \times c'$
      where $v_{last}$ is the last value of partial aggregate sent to the dependent of $n$,
      $c'$ is the coherency requirement on the partial aggregate and
      $\alpha$ is a preselected real number.
    At each epoch
      Update the model parameters for the prediction scheme
      If $lastPushTime - CurrentTime > NoActivityThreshold$
        Push $v_n$ to dependent
        set $lastPushTime = CurrentTime$
```

Fig. 3. Pseudocode for Aggregate computation under AP

Comparison with synchronous approaches:
We would like to point out that our asynchronous approach is in direct contrast with the epoch-based synchronization scheme of TAG [Madden et al. 2002], Cougar [Yao and Gehrke 2002] and TiNA [Sharaf et al. 2003]. In that approach each epoch is divided into time slots and all serving nodes at a given level in the tree are allowed to transmit within a particular time slot. The dependents of these nodes listen during this time slot. At the end of this time slot, each dependent computes a partial aggregate of the data received from its serving nodes. During the next time slot, the dependents transmit the partial aggregates to their dependents. In synchronous computation methods, the duration of the time slot for which the message is withheld at each aggregator node decides the amount of delay for each value that is generated at a source to reach the query node. This delay can lead to loss in fidelity, unacceptable for scenarios that require online decision making. Using an asynchronous approach minimizes this delay, thus providing the potential to deliver higher fidelity. In terms of lifetime, with the asynchronous approach it may seem that computation of an aggregate on receiving a message from any serving node, and a subsequent push to the dependent, if required, may lead to unnecessary transmissions and thus a decrease in lifetime. Our approach of using in-network filtering and in-network prediction for energy efficient aggregation in Section 2.3 ensures that this is not the case. In Section 4.2 we show the superiority – both with respect to fidelity and lifetime of our asynchronous approach over the synchronous approach.

2.3 Avoiding unnecessary partial aggregate propagations

An aggregate query has associated with it a user specified coherency requirement. In this section, we present two ways, *a) In-Network Filtering* and *b) In-Network Prediction*, in which this coherency requirement can be used by nodes of the aggregation tree to conserve energy.

In Figure 2, whenever E computes a new partial aggregate, v_n, it performs in-network filtering whereby it compares v_n with the value last sent to B, (say) v_{last}, and forwards v_n to B only if it finds that the new value of the partial aggregate is *relevant* to B i.e., if the value of the partial aggregate, v_n, differs from the last value, v_{last} sent to the dependent by more than c', the coherency requirement on the partial aggregate.

Further, prediction can be used to conserve energy by transmitting even the relevant partial aggregates only when required. In Figure 2 consider serving node J and dependent B of E. We saw in Section 2.2 that if E does not receive a new partial aggregate from J, it predicts it when computing its own partial aggregate. The same applies to B's actions as well. So, if E estimates that B would correctly predict E's partial aggregate, E does not have to send it to B. After computing a new partial aggregate v_n, a node also computes the value, v_p, of the partial aggregate from E which its dependent node would predict at that instant. The value v_n is propagated to the dependent only if the prediction made by it is deemed *inaccurate*, i.e., when it exceeds a factor α of c'.

To conclude, an aggregator node propagates the computed partial aggregate v_n to its dependent if:

$$|v_n - v_{last}| > c' \tag{1}$$

and

$$|v_n - v_p| > \alpha \times c' \tag{2}$$

where α is a real number. In Section 4.2.4, we determine experimentally, an approximation to the optimal value of α. α controls the degree of error that is considered tolerable by the prediction mechanism. From the above discussions, we note that the factor $\alpha \times \beta$ is applied to the coherency (c) in making a decision whether or not to push a partial aggregate to its dependent.

In order to ensure accurate prediction of partial aggregates, we use the notion of *NoActivityThreshold* to guard against loss of messages (and thus loss of model parameters and partial aggregate values) due to collisions. *NoActivityThreshold* at a node for its dependent node is defined as the amount of time for which a node waits before pushing the value of its partial aggregate to the dependent, i.e. if a node has not pushed the value of the partial aggregate to its dependent for a duration greater than *NoActivityThreshold*, the node pushes the value of the partial aggregate.

2.4 In-Network Prediction of Partial Aggregates

In this section we present two different schemes for in-network prediction of the values of the (partial) aggregates. Section 2.4.1 presents a method for predicting values of unknown partial aggregates by using data *trend*. Section 2.4.2 presents a regression based approach to predicting partial aggregates. Section 2.4.3 explains how the prediction models are maintained. Section 2.4.4 discusses the memory and computational overheads of the in-network prediction schemes.

2.4.1 *Prediction using data trend.*

In most examples of dynamic data for e.g. weather parameters, the average short term behavior can be modeled by a linear function of time. Consider that the value of a data item is available at times t_n, $n = 0, 1, ...$ The value of the data item v_t at time t such that $t_n < t < t_{n+1}$ is estimated by

$$v_t = v_{t_n} + a(t_n) \times (t - t_n) \tag{3}$$

where $a(t_n)$ is the model parameter. $a(t)$ is the rate of change of value of the data item at time t.

Estimating rate of change of data. We adopt the notion of data *trend* from financial time series forecasting. For time varying data v_t, the *trend* of the data at time t is defined as the gradient of its expected value at that instant.

$$a(t) = \frac{d}{dt}(\mathbf{E}(v_t)) \tag{4}$$

To estimate $a(t_n)$, each node maintains a past history of periodically observed values v_t, over a moving window of size W. The past w values at time t_n are v_{t_i}, $i = n - w + 1, n - w + 2, ..., n$. The estimated value of $a(t_n)$, denoted by $\hat{a}(t_n)$, is the average of first differences of values over this moving window.

$$\hat{a}(t_n) = \frac{1}{w} \sum_{i=n-w+1}^{n} (v_{t_i} - v_{t_{i-1}}) \tag{5}$$

2.4.2 *Prediction using Recursive Least Squares.* In this section, we present *Recursive Least Squares*, RLS [Young 1984], an alternative to the prediction scheme presented in Section 2.4.1. Consider a data source (or an aggregator node) that generates a stream of values (or partial aggregates) v_1, v_2,...,v_{n-1} at corresponding instants of time $t = 1, ..., n-1$. To predict the value that would be generated at the next instant of time $t = n$, we use linear regression to obtain a least squares fit for the data. The predicted value of data at time $t = n$, denoted $\hat{v}_n$ is expressed as a linear function of a window of past w values as follows: $\hat{v}_n = a_1 v_{n-1} + a_2 v_{n-2} + ... + a_w v_{n-w}$. The coefficients a_i for $i = 1, ..., w$ are called model coefficients (or model parameters). At $t = m + w$, considering m successive sliding windows each of size w gives m equations. In the matrix notation, these equations are represented as:

$$V \times a = y \tag{6}$$

$$where \;\; V = \begin{bmatrix} v_1 & v_2 & ... & v_w \\ v_2 & v_3 & ... & v_{1+w} \\ ... & ... & ... & ... \\ v_m & v_{m-1} & ... & v_{m+w-1} \end{bmatrix} \tag{7}$$

$$a = \begin{bmatrix} a_1 \\ a_2 \\ ... \\ a_w \end{bmatrix} \; and \; y = \begin{bmatrix} v_{1+w} \\ v_{2+w} \\ ... \\ v_{m+w} \end{bmatrix} \tag{8}$$

The model coefficients at $t = m + w$ that satisfy this equation are those that minimize the sum of the squares of the error $e(t)$ for $t = 1 + w, 2 + w, ..., m + w$ given by:

$$\sum_{t=1+w}^{m+w} (v_t - \hat{v}_t)^2$$

That is, we want to find a such that it minimizes the objective:

$$(V \times a - y)^T \times (V \times a - y)$$

where X^T denotes the transpose of matrix X. The model coefficients are given by:

$$a = (V^T \times V)^{-1} \times (V^T \times y) \tag{9}$$

Computing the model parameters. Since all data values are not available before-hand, we compute the model parameters using a technique called Recursive Least Squares (RLS). Let $a(m + w)$ be the estimated model parameters at time $t = m + w$. When a new data item is available, this new value is used to recompute the model parameters. If we denote by P_{m+w}, the value $(V^T \times V)^{-1}$ at $t = m + w$, we can write Equation 9 as:

$$a(m + w) = P_{m+w} \times (V^T \times y) \tag{10}$$

When the data item v_{m+w+1} is received, by applying the matrix-inversion lemma, it can be shown that the model parameters can be efficiently updated using the following set of equations:

$$\begin{aligned} P_{m+w+1} &= P_{m+w} - P_{m+w} \times X_{new} \times (1 + X_{new}^T \times P_{m+w} \times X_{new})^{-1} \\ &\quad \times X_{new}^T \times P_{m+w} \\ a(m + w + 1) &= a(m + w) - P_{m+w+1} \times (X_{new} \times X_{new}^T \times a(m + w) - X_{new} \times v_{m+w+1}) \end{aligned}$$

where X_{new} is the row vector containing values v_{m+1}, v_{m+2}, ..., v_{m+w}.

2.4.3 *Maintaining and updating parameters of the prediction models.* To enable prediction, each node of the aggregation tree maintains:

— The current estimated model parameter(s), $a(t)$, of the data that it pushes up the aggregation tree. This is used to compute the value that the dependent would predict at any given instant. The trend based scheme uses a single model parameter, data trend, of size 4 bytes. The number of model parameters that can be used for RLS-based prediction is dictated by the value of w. Since these parameters are sent to the dependent and the messages sizes are small, w needs to be small. In general it is possible to use a value of w so that transmission of model parameters does not require any extra messages.

— The value of $a(t)$ that it receives along with the partial aggregates sent by each of the serving nodes. This is used by the node to predict the value of the partial aggregate at instants between two successive receptions of partial aggregates from the serving node.

In order to keep the parameters of the prediction model, $a(t)$, up-to-date, whenever a node estimates the model parameters, it checks for the validity of conditions in Equations (1) and (2). If both the conditions are satisfied, it sends to its dependent the current value of its partial aggregate along with its current model parameters. To compute the current value of its partial aggregate, a node uses the most recent values of the model parameters of the partial aggregates at its serving nodes.

2.4.4 *Memory and computational overheads of prediction.* We now analyze the memory and computational overheads of the prediction schemes proposed in Sections 2.4.1 and 2.4.2.

Trend-based scheme:

The trend-based scheme requires each node on the aggregation tree to maintain:

(1) An array of size w containing first differences of the partial aggregate it sends to its dependent sensor. This would require $4w$ bytes, since each float value requires 4 bytes of memory. This array is used in calculating the *trend* parameter, that needs to be sent along with the actual partial aggregate.

(2) Two float values to store the last value of the partial aggregate and last value of *trend* sent to the dependent. This requires 8 bytes.

(3) Corresponding to each serving node, the last received value and the *trend* parameter for each serving node is maintained. If we denote by n_p, the number of serving nodes for a given node, the amount of space required is $4 \times 2 \times n_p$ bytes.

Thus, the total amount of space overhead is $4 \times n_p \times 2 + 4 \times w + 8$ bytes.

Each time a new value is received from a serving node, a node predicts the value of the partial aggregator at the other serving nodes and its dependent sensor. Each prediction requires a single multiplication and an addition operation. The number of operations required is $4 \times n_p \times 2$. At each epoch, to update the moving window of its own values, it requires n_p arithmetic operations and an array lookup. To re-estimate its model parameters, when the *trend* parameter is re-estimated, we require one multiplication, one addition and one division operations. It is clear that the computation overhead associated with this scheme is reasonable even for the most resource constrained sensor nodes.

For example, in our simulation setup, each node has an average of 4 serving nodes, and a value of $w = 20$, thus translating to a space overhead 120 bytes per sensor and computation overhead of 7 arithmetic operations at each epoch.

RLS-based scheme:
The RLS-based scheme maintains w model parameters each of size 4 bytes and the last w values of the partial aggregate received from each of its serving nodes. Also, w model parameters are maintained for the partial aggregate sent to the dependent sensor. The P matrix in Equation 10 requires a constant space of $w^2 \times 4$ bytes. Thus, each node incurs a memory overhead of $4 \times (n_p \times w \times 2 + w \times 2 + w^2)$ bytes.

For each message received from a serving node, to compute the predicted value of the aggregate at the dependent and at the other serving nodes, $2 \times w - 1$ operations are required. A total of $n_p \times (2 \times w - 1)$ operations are thus required. At each epoch, to re-estimate the model parameters, approximately $8 \times w^2 + 3 \times w$ arithmetic operations are required. To compute the value of partial aggregate at that epoch, for each serving node from which no value has been received, w multiplications and $w - 1$ additions are required. In our simulation setup, w for RLS is 4, $n_p = 4$, thus leading to a space overhead of 224 bytes and computation overhead of at least 140 arithmetic operations at each epoch. In general, we observe that the amount of computation required to perform in-network prediction using RLS-based method is higher than that required for the trend-based method.

In this section, assuming that an aggregation tree already exists, we proposed AP for efficient computation of the value of the aggregate by minimizing the number of message transmissions. The next section discusses our tree construction algorithm.

3. AGGREGATION TREE CONSTRUCTION

An aggregation tree containing the source nodes is constructed as described in this section for aggregate computation with the query node as the root. For simplicity of explanation, we assume that the user requests for an aggregate of values sensed by sensors in a target region by specifying the center of the target region and the coordinates of the smallest bounding rectangle that contains the target region. An aggregation tree should have the following properties:

(1) Changes to data sensed at the sources should be transmitted to the query node with minimum communication delays for the longest possible duration of time.

(2) An aggregate of the data sensed at the sources should be transmitted to the query node using the minimum possible number of messages.

In order to achieve these objectives, we first construct a *backbone path* from a source near the center of the target region to the query node as explained in Section 3.1. The nodes on the backbone path are chosen in such a manner that given the initial energy of the sensors, it is the best possible path. Once the backbone path has been constructed, the other nodes in the target region join the aggregation tree. In constructing the rest of the tree, we attempt to maximize the message transmission savings provided by in-network aggregation by doing it as close to the sources as possible. To do this, at each step, the sources in the target region that are not a part of the aggregation tree join the tree constructed thus far by finding a path to a node on it. To determine this node, we note that finding a path to any node on the tree is equivalent to finding a path to the query node due to the possibility of in-network aggregation at these nodes. This helps us achieve the objective of minimizing the number of message transmissions. Section 3.2 explains the construction of dissemination paths from the rest of the nodes in the target region to the query node. Section 3.3 describes our optimization to handle multiple aggregate queries efficiently. In Section 3.4, we explain how user specified coherency requirements are incorporated into the aggregation tree. Section 3.5 describes different choices of preference criteria for choosing nodes on the dissemination paths. Section 3.6 describes modifications to the tree construction algorithm to handle complex aggregate queries with `group by` clauses. We present our approach to handle failure of nodes on the aggregation tree in Section 3.7.

3.1 Backbone path construction

We now explain the construction of the backbone path from the query node to a sensor, called *backbone source*, which can sense the region near the center of the target region. A request for the value of an aggregate that is injected at a query node is diffused into the network by broadcasting a REQUEST message to all its neighbors. The backbone source sends a RESPONSE message, advertising its ability to serve the requested data, after a certain time interval. Notice that there may be multiple such nodes. If the data is unavailable with the sensor, it forwards the request to its neighbors. Thus the original request is diffused throughout the network. Every sensor that forwards the request to its neighbors waits for some time for their responses. As the initial request message gets diffused into the network this time-out value *monotonically decreases* with the number of hops away from the query node.

Upon time-out, if a node receives at least one RESPONSE, it chooses the *best* of these responses based on a *preference factor*, PF, detailed in Section 3.5 and broadcasts a RESPONSE. Finally, the query node gets responses from one or more of its neighbors and chooses the *best* response from the set of received responses.

After selecting the best response, the query node considers the sender of that response as its serving node and sends it a SEND_UPDATES message requesting the setup of a path to serve updates to the requested data. A sensor A that receives a SEND_UPDATES message from sensor B starts serving data if it is already available with it, otherwise it forwards the message to the sender of the best response. In both cases, sensor A adds sensor B as its dependent for that data item. In this manner, a backbone path is established from a source of the data item at the center of the target region to the query node.

3.2 Construction of rest of the tree

The backbone path constructed using the method described in Section 3.1 is used for setting up the aggregation tree as follows: When the backbone source receives a SEND_UPDATES message, it not only starts disseminating updates (to the sensed data) to the query node, but also broadcasts an AGGR_REQUEST message requesting its neighbors in the target region to join the aggregation tree. Nodes in the target region advertise their ability to serve the requested data by broadcasting an AGGR_RESPONSE message. On receiving an AGGR_RESPONSE, a node treats it in the same way as a RESPONSE message, except if it already lies on the aggregation tree constructed so far. In the latter case it forwards the AGGR_RESPONSE to its dependent up the aggregation tree. This is justified because a node on the aggregation tree already has a path towards the query node along which it can propagate the partial aggregates computed by in-network aggregation. When the query node receives this AGGR_RESPONSE, it sends a SEND_UPDATES message towards the source from which the AGGR_RESPONSE originated. The new source node joins the tree in a manner similar to the backbone source. It then broadcasts an AGGR_REQUEST and the above set of events repeat themselves so that more nodes within the target region can join the aggregation tree. During the construction of the aggregation tree, messages exchanged between nodes (for eg. REQUEST, RESPONSE, AGGR_REQUEST, AGGR_RESPONSE, SEND_UPDATES) may be lost. In order to deal with loss of these messages (due to collisions), we transmit each message multiple times. In our experiments we observe that transmitting messages upto a maximum of three times ensures successful construction of aggregation trees. We conclude that this is due the redundant message transmission paths created by the broadcast of messages along with retransmissions.

Figure 4 presents an illustration of the tree construction algorithm. In Fig. 4 (a), the "Query Node" requests for the aggregate of data sensed in "Target region" by sending a REQUEST message. Node A responds to this request by broadcasting a RESPONSE message. This response message is broadcast till it reaches the query node. This is shown in 4 (b). The query node creates and sends a SEND_UPDATES message to its best serving node as shown in (c). This node is propagated till it reaches node A. On receiving the SEND_UPDATES message, A starts transmitting the data it senses in an AGGREGATE message. A also broadcasts an AGGR_REQUEST message, requesting its neighbors to join the aggregation tree formed thus far. This is shown in 4 (d). Node C on receiving the AGGR_REQUEST, sends an AGGR_RESPONSE indicating that it is one of the source nodes. The response is broadcast till it reaches node F. Node F being a part of the aggregation tree, already knows a path towards the query node. Thus, node F does not broadcast the AGGR_RESPONSE but sends it to its dependent node. The dependent node in turn sends it to its own dependent, till it reaches the query node. The query node, sends a SEND_UPDATES message requesting C to start serving data. F, it may be noted is a node where in-network aggregation takes place.

It might seem that the aggregation tree construction algorithm incurs a high overhead in terms of the number of messages transmitted. However, we believe that the aggregation trees constructed using this mechanism perform much better during aggregate dissemination than those constructed by ad hoc tree construction algorithms. Aggregation trees constructed by our scheme give higher lifetime and lower fidelity losses than those created by ad hoc tree construction algorithms.

In the tree construction algorithm, a source node that is not reachable from the backbone source never receives an AGGR_REQUEST message. Also, it is possible that there is no node near the center of the target region. In order to enable

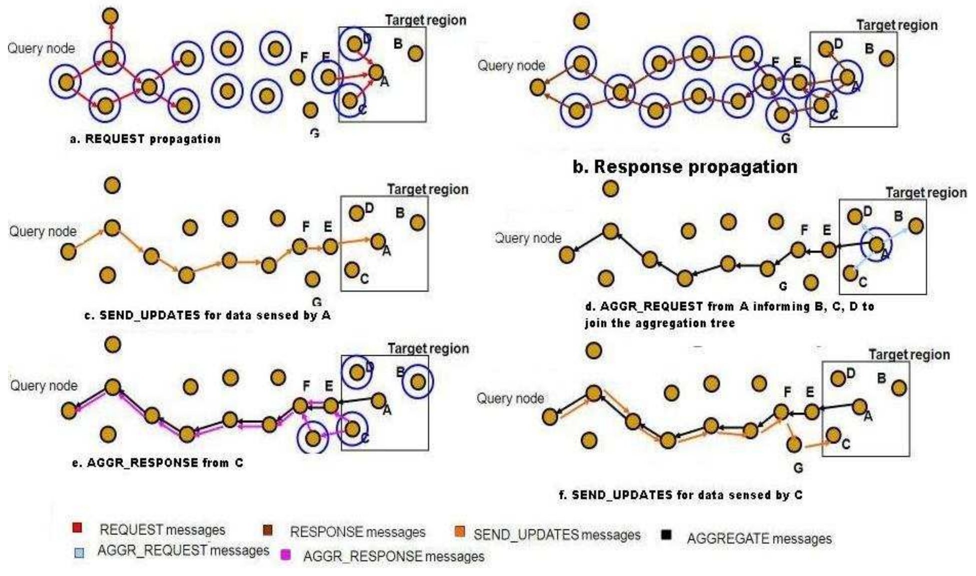


Fig. 4. Illustration of Tree construction

such nodes to join the aggregation tree, after receiving the REQUEST message for the aggregate query, each source node in the target region chooses a random interval of time to wait. When the random interval expires, a source node initiates the construction of a dissemination path towards the query node by sending a AGGR_RESPONSE message, if none of the following conditions hold:

(1) It is already on the aggregation tree.
(2) It has received an AGGR_REQUEST message. This means that this node is reachable from another source node in the target region and that the construction of dissemination path from the node to the query node has already been initiated.
(3) It has received either a RESPONSE or an AGGR_RESPONSE message from another source node. This means that a node from which this node is reachable has already initiated the dissemination path construction and that an AGGR_REQUEST will be received in the future. Thus, each node eventually initiates the construction of a dissemination path towards the query node.

It is to be noted that using the above mechanism, theoretically multiple backbone paths are possible for the case where there is no node near the center of the target region. However, we observe in our experiments that this never happens.

3.3 Optimization across Multiple Aggregate Queries

In this section we propose Multi-Query Optimization (*MQO*), an extension to our tree construction algorithm to efficiently handle multiple aggregate queries with overlapping target regions. Consider for instance two queries, one of which requests for an aggregate of values sensed by sensors on the first floor of a building, while the other query requests for the same aggregate over the values sensed by sensors on the southern wall of the building. These queries share a common region which consists of those sensors on the southern wall of the first floor of the building.

We reuse parts of the aggregation trees (and thus data available with the nodes of already established queries) during tree construction for a new query. Suppose that there are two queries Q_1 and Q_2, and the target regions for these are T_1 and T_2 respectively. Let the region common to T_1 and T_2 be T_c, which contains a non-zero number of sensor nodes. In order to identify maximal common subtrees which can be shared across both queries, we find those nodes on the aggregation tree of Q_1 where the partial aggregate pushed to the dependent consists of values sensed by the nodes only from the common area T_c. We say that partial aggregate of values sensed by a set of sources $S_1, S_2, .., S_k$ in the common region T_c *loses its identity* for query Q_2 at some common ancestor node N_a on the aggregation tree for query Q_1 if the subtree rooted at N_a contains at least one source node of Q_1 that does not belong to T_c and no other node of which N_a is an ancestor satisfies this property. If data does not lose identity at node N_a, partial aggregates received by it for query Q_1 can also be used to serve query Q_2.

Assume that Q_1 has already been set up and Q_2 is now injected at the query node. For simplicity of exposition, assume that the backbone source of Q_2 does not belong to the common target region T_c. The case where the backbone source belongs to the common region can be easily handled. The backbone path for Q_2 is formed using the mechanism described in Section 3.1. The backbone source then broadcasts an AGGR_REQUEST. Let S_c be a node in the common region that receives an AGGR_REQUEST. S_c initiates a search for the root of the largest subtree (of the aggregation tree for Q_1) containing it and all of whose nodes belong to the target region of Q_2. Note that this node is the child of the node at which the partial aggregate loses identity. In order to search for the root of a common subtree, S_c sends a message of type AGGR_EXPLORE to its dependent in the aggregation tree for Q_1. The recipient of this message, forwards the message to its dependent if the partial aggregate that it is being served by S_c does not lose identity for Q_2. If the partial aggregate loses identity the node broadcasts an AGGR_RESPONSE message if it has not already sent an AGGR_RESPONSE for Q_2 (in response to an AGGR_EXPLORE from a different source in the common target region), indicating its ability to serve a partial aggregate required by the new query, Q_2. We call this node a *pseudo source* for the partial aggregate containing the values sensed by source S_c. The AGGR_RESPONSE is handled in the same manner as explained in Section 3.2. Subsequently, the pseudo source receives a SEND_UPDATES message requesting the node to send updates to the partial aggregate. Source nodes of Q_2 that are not common to Q_1 join the aggregation tree for Q_2 in the manner described in Section 3.2.

3.4 Incorporating coherency requirements into the tree construction algorithm

In general a sensor node can be on the aggregation trees of more than one aggregation query with different dependents having different coherency requirements. The coherency of partial aggregates available at a sensor is such that it is able to serve all its dependents at the desired coherency requirements. Given the above tree construction algorithm, how are user specified coherency requirements handled in the aggregation tree construction? Any sensor B, while sending the SEND_UPDATES message to a sensor A also specifies a desired coherency value, c_B. If the partial aggregate is already available at sensor A at coherency c_A and if $c_A > c_B$, i.e., c_A is looser than c_B then sensor A requests all its parent sensors that lead to nodes in the target region of this query to tighten the desired coherency for partial aggregate received by A to c_B else if $c_A < c_B$ the request is satisfied.

3.5 Preference Factor – to determine the best data provider for a node on a path

Serving nodes along a dissemination path are chosen on the basis of some *Preference Factor*, PF. In this subsection we present three different criteria that can be used to find the best sensor among those that respond positively (with a RESPONSE message during construction of backbone path or AGGR_RESPONSE message during the construction of the rest of the aggregation tree):

(1) Energy (PF-e): With PF-e a node B chooses a node A as its serving node, if A has the smallest number of hops, e, between it and the source of data. This is because the energy e expended along a path is directly dependent on the number of hops in the path.

(2) Remaining Lifetime (PF-l): With PF-l, a node chooses that node which is estimated to have the highest remaining lifetime among the possible paths from from a data source to the chosen sensor and onto the sink. The lifetime of a path between two sensors is the smallest of the remaining lifetimes of the sensors along the path. The lifetime l of a subtree in the aggregation tree is the smallest of the lifetime of the dissemination paths that constitute the subtree. After the aggregation tree is setup, dependents are periodically updated with the remaining subtree lifetime, by piggybacking lifetime information on AGGREGATE messages.

(3) PF-el : PF-el prefers responses with low values of e and high values of l. Thus, PF-el selects the responses with the least value of $PF = \frac{e}{l}$.

3.6 Complex Aggregate Queries

Often, instead of a single aggregate over data sensed by sensors, aggregates over sets of regions are required. One such typical query would be: "Report the average of the temperature sensed by sensors on each wall of the fourth floor of the building whenever the average changes by 1^oC". Such type of queries are called `group by` queries. Each wall constitutes a group in this query. In such cases, each node on the dissemination graph of this query potentially needs to send and receive one partial aggregate per group in every epoch. In the following discussion we assume that no more than one partial aggregate can be encapsulated in a message. In this section, we look at modifications to the tree construction algorithm to efficiently handle `group by` queries. As in the case of simple aggregate queries, if a dissemination tree is constructed, each sensor on the tree may be required to receive a partial aggregate value for each group in every epoch. Instead, we note that it may be energy efficient to have each partial aggregator node route data for different groups to different dependents in which case a dependent node is required to receive a partial aggregate value only for a subset of groups in every epoch. In order to achieve this, we propose two modifications to the tree construction algorithm:

(1) We divide the query into subqueries based on the grouping criterion specified by the query. In the above example, one subquery would be "Report the average of the temperature sensed by sensors on the eastern wall of the fourth floor of the building whenever it changes by 1^oC". We create an aggregation tree for each subquery using the tree construction algorithm in Section 3. Thus, the dissemination structure is no longer a tree; it is a dag (directed acyclic graph) of (possibly) intersecting trees.

(2) The number of different groups whose aggregation trees intersect at a node is an indicator of the number of partial aggregate messages that the node would have to send in the worst case (one per group in every epoch). To distribute this communication overhead across nodes in the network, we modify the preference criterion for the choice of nodes on the aggregation tree as follows: Whenever a AGGR_RESPONSE message is sent by any node, it sends the current number of aggregation trees intersecting at that node, n_g. The recipient of an AGGR_RESPONSE chooses as its serving node that node which sent the response with the best value for the preference criterion. In the case of PF-el, the new preference factor is given by:

$$PF = \frac{e \times n_g}{l}$$

where e and l have the same meaning as in Section 3.5.

3.7 Recovery from failures

In this section, we explain how our tree construction scheme can recover from node failures. Failure of an aggregator node leads to loss of a whole subtree of updates. We deal with such failures by initiating a recovery process that finds an alternate path from each node serving a failed node to some ancestor (or a sibling of some ancestor) of the failed node.

Any sensor whose energy falls below a certain threshold, broadcasts a DEATH message into the network. On receiving the DEATH message, nodes on the aggregation tree that were serving the failed node consider themselves pseudo sources and broadcast AGGR_RESPONSE messages for each aggregation tree of which they are a part. These AGGR_RESPONSEs are ignored by nodes in the tree that are at a level higher (away from the query node) than the pseudo sources. Nodes in the tree that are at a level lower than the pseudo sources process the message like in the case of the tree construction in Section 3.2. The pseudo source nodes eventually receive a SEND_UPDATES message for each query for which they broadcast a AGGR_RESPONSE message.

In order to handle random failures, each node periodically broadcasts an ALIVE message, advertising that it is alive. When a serving node does not receive this message from its parent at least once in several successive intervals, it concludes that the parent has failed and initiates a path repair. In this fashion, each disconnected serving node finds a path back to the query node.

4. EXPERIMENTAL EVALUATION

This section gives the details of the experimental setup and results for various studies. For our simulations, we use TOSSIM [Levis et al. 2003], simulator for TinyOS

[Hill et al. 2000], a sensor network operating system. TOSSIM models the 40Kbit RFM mica [Hill and Mica 2002] networking stack, including the MAC, encoding, timing and synchronous ACKs. Radio contention is modelled by the simulator using a CSMA based mechanism. Loss of messages due to collisions, hidden-terminal problem and arbitrary bit errors are taken into account. TOSSIM represents the wireless network as a directed graph, with each sensor as a node, and each edge having an associated bit error probability. By replacing a few low-level components of TinyOS, TOSSIM translates hardware interrupts into discrete simulator events; the simulator has an event queue which then delivers the interrupts that drive the execution of the TinyOS application. Consequently, any code that runs on TOSSIM can be easily transitioned to real sensors. Details of the ADC and radio models, such as readings and loss rates, can be both queried and set. Programs can also receive higher level information, such as packet transmissions and receptions or application-level events. We use TinyOS version 1.0 in our simulations. Complete details may be found at [TinyOS Website].

4.1 Experimental Setup

Data Traces: The performance of all our schemes was studied using traces of real world sensor data. The results presented here are based on the sensor data traces collected on-board different ships which belong to *GLOBEC Georges Bank Cruises*[2] [Inventory of U.S. Globec Georges Bank Data, Project: AL9508 1995]. We experimented with two different types of data: (a) data streams sensed at the sources are correlated. In order to generate data traces sensed at different sources, a gaussian noise with mean 0 and variance 25 was added to the base data trace plotted in Figure 5 and (b) data streams sensed at the sources are uncorrelated, as described in Section 4.2.2. Though this data is sensed once every minute, for experimentations, the updates are considered to occur once in every epoch (See Table II). Note that the data was scaled and offset so as to fit into the range of values that the ADC interface generates (16 bit unsigned integer), and to provide sufficient leeway to set coherency requirements to different values for simulation purposes.

Energy Model: Energy consumed by a sensor from the beginning of its operation, is calculated considering idle time power dissipation ($5mW$), receive power dissipation ($50mW$) and transmit power dissipation ($100mW$). These values are based on the ratios for these values shown in [Kaiser] and used in [Intanagonwiwat et al. 2000]. Considering that the sensor spends r seconds in receiving messages and t seconds in transmitting messages and is idle for i seconds, the total energy consumed by that sensor, κ, is calculated as:

$$\kappa = (r \times 50) + (t \times 100) + (i \times 5)$$

Since computations related to in-network filtering and in-network prediction happen only for a short duration of time each time a new data value is sensed or received, the energy expended in such computations may be ignored.

[2]We also conducted experiments using other data traces such as stock ticker values for sensed data. Results indicate that performance will only be better than those with the air temperature data traces.

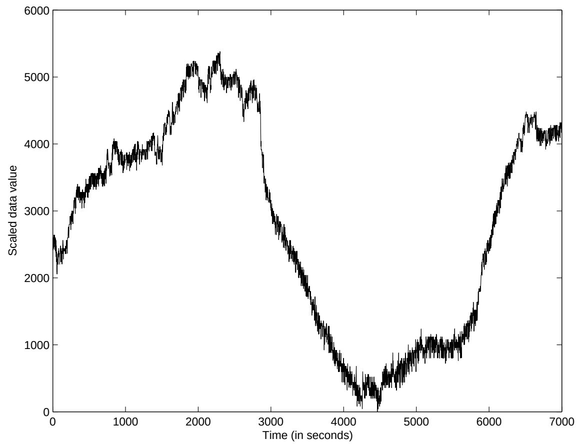


Fig. 5. Plot of base data trace used for generating the correlated data streams.

In TOSSIM there is no way for a sensor to find out its remaining battery power. To overcome this, a variable in the state of every sensor is used to hold the energy available with the sensor. The available energy of the sensor is decremented on each send or reception of message according to the above equation. The energy of the sensor is decremented for the time that it has been idle.

Terrain size	$100m \times 100m$
Transmission range	10m
Initial Energy	100K units
Number of sensors used	200
Number of sources per query	35
Target Area	$10m \times 100m$
Epoch Duration	2 sec
Coherency on value of aggregate[1] (c)	125, 200, 250, 375, 500
Coherency on value of partial aggregate (c')	$c \times 0.2$ (see Section 4.2.3)
α	2.0 (see Section 4.2.4)
Preference Criterion	PF-el (see Section 4.2.5)
Prediction mechanism	data trend with w = 20 (see Section 4.2.7)[2]
NoActivityThreshold	$25 \times EpochDuration$

Table II. Nominal Simulation Parameters

A rectangular grid topology was used for the sensor network. We use a packet size of 36 bytes, with a 29 byte data payload. Unless specified otherwise, for an experiment, simulation parameters are set as in Table II.

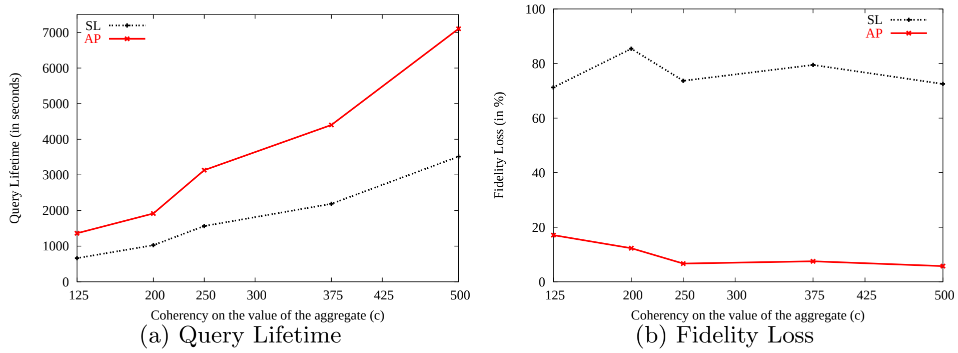
[1]Note that a value of 250 corresponds approximately to 1^oC in the unscaled data
[2]Our experiments show that even with higher values of w the performance of our algorithms is the same.

4.2 Experimental Results

In Sections 4.2.1 through 4.2.11 we assume that the aggregate to be computed is the average of values sensed by the nodes in the target region. An AGGREGATE message sent by any node N, to its dependent, for average computation contains a pair *(sum, count)*, where *count* is the number of sources that belong to the subtree rooted at N and *sum* is the sum of values sensed by them. The partial aggregate computed by N is $sum/count$. In Section 4.2.12 we evaluate our scheme for average queries with `group by` clauses. In Section 4.2.13 we present results to demonstrate the applicability of our scheme for other types of aggregates – average, maximum and min queries as well. We define *lifetime* as the amount of time for which the query node receives updates to the value of aggregate from majority, i.e., at least 50%, of sensor nodes in the target region. We have evaluated the lifetime and fidelity loss for percentages other than 50% and observe that the results follow a similar trend.

Calculation of Fidelity Loss: In our experiments we calculate fidelity loss as follows: Suppose a node senses a new value at time t_1. The new value of the aggregate at t_1 is calculated. If $|a_{new} - a_{old}| > c$, where c is the coherency on the value of the aggregate, this means that the value of the aggregate known at the query node is not within allowable limits of the actual value of the aggregate. We compute the portion of time during the lifetime of the query for which this is true, and call this *Infidel time*. Fidelity Loss for the aggregate query is computed as

$$Fidelity\ Loss = \frac{Infidel\ time}{Lifetime}$$



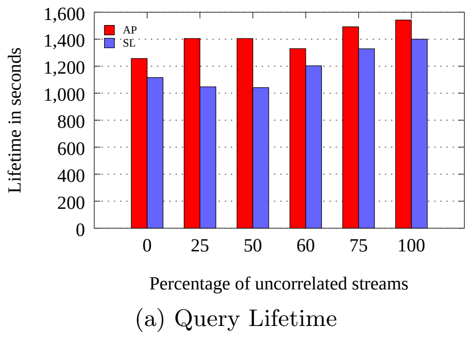
(a) Query Lifetime (b) Fidelity Loss

Fig. 6. Comparison of AP and SL based aggregation schemes

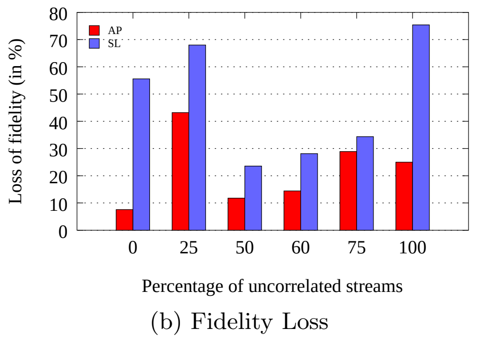
4.2.1 *Asynchronous aggregation + prediction is a winning combination.* This section presents a comparison of experiments using AP based aggregation and SL based technique on correlated data streams. Figure 6 shows the query lifetime and fidelity loss comparison for different values of user specified coherency requirements. From the figure, AP produces a 8% fidelity loss on the average over all coherency requirements whereas SL's loss is close to 80%. This is because the synchronous scheme delays each message for some amount of time at partial aggregator nodes before transmitting the partial aggregate to the dependent. Thus, an update to the value sensed at a source is not propagated to the query node immediately, thus leading to loss of fidelity. In terms of lifetime, AP gives higher lifetimes than SL. The difference between the query lifetimes obtained using SL and AP increases as the user specified coherency gets looser. The number of messages that are filtered by AP through the condition in Equation 2 in Section 2.3 increases with loosening of coherency requirement corresponding to a decrease in number of messages sent. Overall, AP gives an 80% improvement in lifetime over the SL based aggregation scheme. This may be attributed to the message transmission savings obtained by avoiding transmissions by predicting the value of data known to the dependents. We observe that aggregation using AP injects at least 30% lower number of messages into the network per second than SL. This explains the higher lifetime.

AP has two components in it – *a*) asynchronous computation and *b*) in-network prediction of partial aggregates. We study the performance of each of these components separately to find out which of these two contributes more towards the better performance of AP. To this end, we use the last received value (instead of the current value of partial aggregate at serving nodes) and synchronous aggregate computation from SL and implement *asynchronous last-value* (AL) and *synchronous prediction* (SP) based aggregation schemes respectively. Table I compares these in terms of the fidelities and query lifetimes that they deliver. For instance, for user specified coherency of 250, SL, SP, AL, AP give fidelity losses of 73.67%, 20.68%, 3.2%, and 6.66% respectively. The corresponding query lifetimes in seconds are 1564, 2462, 1209 and 3132. SL gives higher lifetime than AL due to synchronization of message transmissions. This, as said earlier, introduces delays in aggregate propagation thus leading to its lower fidelity. SL thus compromises on result quality to achieve higher lifetime. SP synchronizes computation of aggregates at each level on the aggregation tree. The loss in fidelity that is introduced by synchronization is offset by the prediction component leading to higher fidelity than SL. AP is the best choice, giving high fidelity without compromising on lifetime. We note that while AL and AP give comparable fidelity losses, it might be possible to devise an adaptive asynchronous mechanism that chooses between the last value and the predicted value. We leave an exploration of this approach to future work. It might appear that in asynchronous mode, more energy could be consumed due to idle listening at the MAC layer leading to poor performance in comparison with a synchronous aggregate computation mechanism. Since the energy model used in our experiments takes into consideration the energy expended by a sensor during idle listening, we observe that the longer lifetimes are due to the use of prediction in computing the values of the partial aggregates. We thus conclude that AP – asynchronous computation coupled with in-network prediction of partial aggregates gives the best performance. The results also confirm that neither asynchronous computation nor in-network prediction is by itself sufficient in providing both high fidelity and high lifetime – their combination is a must.

4.2.2 *Asynchronous prediction-based aggregation is a winning combination even for uncorrelated data.* In this section we present results for experiments with uncorrelated data streams. For this purpose, we have taken source data from 20

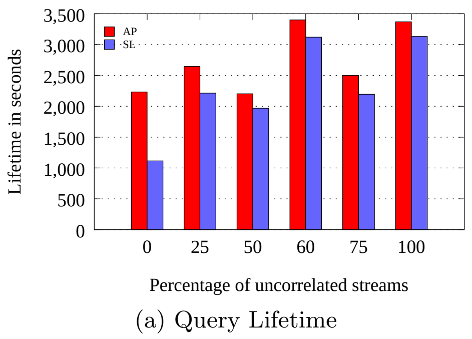


(a) Query Lifetime

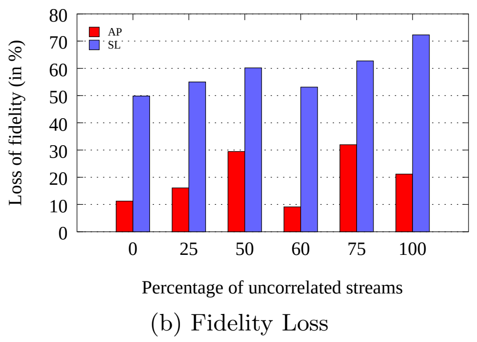


(b) Fidelity Loss

Fig. 7. Results on uncorrelated data for aggregate queries with low coherency



(a) Query Lifetime



(b) Fidelity Loss

Fig. 8. Results on uncorrelated data for aggregate queries with high coherency

different temperature streams from the various excursions of cruise ships in [Inventory of U.S. Globec Georges Bank Data, Project: AL9508 1995]. This data being independent is uncorrelated. We investigate the performance of our scheme when different fractions of the total number of sensors in the target region have data which are uncorrelated, thus exploring the entire spectrum ranging from a scenario with totally uncorrelated data streams to one where certain regions exhibit spatial correlation in data and certain other regions are independent of these. For high as well as low values of user specified coherency, we experiment with different fractions of the total number of source streams that use uncorrelated data (the remaining data streams are generated as specified in Section 4.1). From figures[3] 7 and 8, we observe that AP outperforms SL, giving higher lifetime and better fidelities. As with correlated data, we observe high fidelity losses for SL. AP manages to save on message transmissions by virtue of the prediction scheme, and hence achieves greater lifetime than SL. Since the trend for results on uncorrelated data streams is the same as those on correlated data streams, we use correlated data streams for all further experiments.

[3]Notice that given a particular value of user specified coherency, the trend across different fractions of the target region having uncorrelated data streams is not of significance, since in each of the cases, the data being sensed is different.

4.2.3 *Determination of β.* In Section 2 we defined c', the coherency on the partial aggregate as:

$$c' = c \times \beta$$

where c is the user specified coherency on the aggregate query. Here we determine the value of β that is used in our simulations. Figure 9 shows the variation of lifetime and fidelity loss for different values of β for aggregate coherency of 250. It can be seen that as β increases, fidelity losses as well as lifetime for both AP and SL increase. For any value of β, AP performs better than SL. However, $\beta = 0.2$ gives values of fidelities that are practically useful while also giving high lifetimes. We use this value in the rest of our experiments.

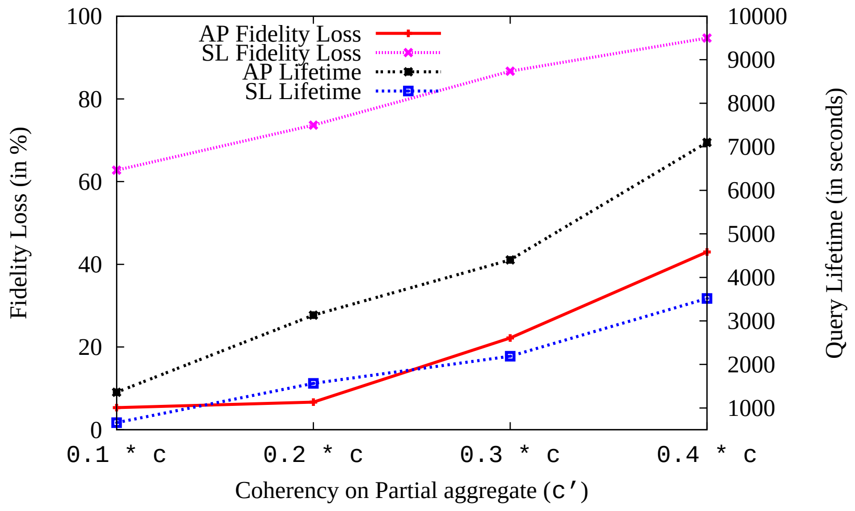


Fig. 9. Variation of Fidelity Loss and Query Lifetime with β

4.2.4 *Determination of α parameter for prediction.* Recall that α signifies that factor of coherency requirement by which the estimated value of aggregate computed by the dependent is allowed to differ from the value of partial aggregate computed by an aggregator node. Figure 10 shows the variation in lifetime and fidelity values for various values of the parameter α for user specified aggregate coherency, $c = 250$. For low values of α, the fidelity loss for the results received at the query node is very low, about 2%. Loss of fidelity increases sharply when α increases above 2.0. Lifetime increases almost linearly with increase in the value of α for $\alpha > 1.0$. From Section 2.3, it is clear that values of the partial aggregate at two nodes which are one hop away from each other in the aggregation tree can differ by at most $\alpha \times c'$ where c' is the coherency requirement on the partial aggregate. This is because the aggregate is pushed only if it differs from the last value sent by more than c' and in addition, it differs from the predicted value by more than $\alpha \times c'$. Thus, as α increases the allowed deviation from actual value increases, leading to lower fidelity. Also, with increase in α the number of partial aggregates that are propagated up the tree decreases, energy consumption is reduced and lifetime increases. Results for other values of c are similar. We conclude that for our experimental setup the optimal value of α parameter is 2.0. In our setup, the average height of the aggregation tree is 22, with each node having an in-degree of 3 (visualize the tree as a dag with edges indicated by the flow of data from sources to the query node). We believe that the optimal value of α depends on the height of the aggregation tree, however we leave a detailed study of this dependence to future work.

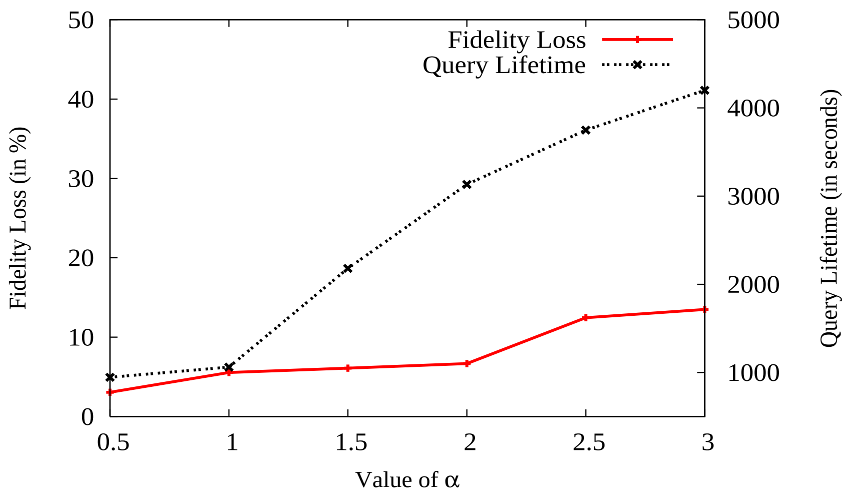
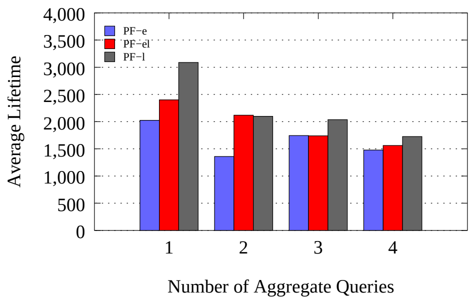
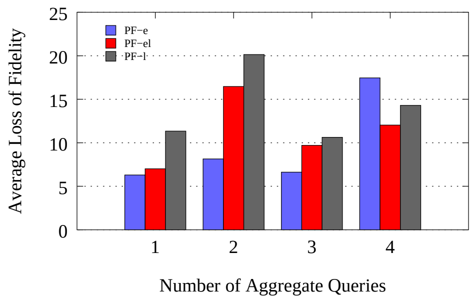


Fig. 10. Variation of Fidelity Loss and Query Lifetime with α

4.2.5 *Preference Factor that combines energy usage and remaining lifetime works best.* We now compare the performance of the different preference criteria for choosing nodes on the aggregation tree presented in Section 3.5. We inject upto a maximum of 4 aggregate queries into the network. Each aggregate query requires the average of values sensed by 16 nodes in a target region. Results of our experiments are shown in Figure 11. The graphs show the variation in *Average Fidelity Loss* and *Average Lifetime* for different number of queries. From the graphs, it can be seen that PF-l maximizes lifetime and may set-up long paths using under-utilized nodes and as a result has a higher loss in fidelity due to higher probability of packet loss. Note that though the energy consumed by PF-l may be much higher, it still maximizes lifetime since it utilizes the lifetime of all the sensors in the network to the maximum possible extent, while PF-e, in an attempt to minimize energy consumption, overuses some sensors leaving others under-utilized. PF-el considers both the remaining lifetime and communication delays and this ensures that it gets better lifetime compared to approaches like PF-e and only slightly lower lifetime than PF-l. We observe that for higher number of queries, PF-e shows a high loss in fidelity due to overloading of certain nodes, thus leading to loss of messages due to collisions. In general, PF-el delivers lower losses in fidelity than PF-l and fidelity losses comparable to PF-e. Clearly, PF-el demonstrates the best tradeoff between lifetime and data fidelity. We conclude that PF-el is the best criterion to choose nodes on the dissemination tree and use it in all our experiments.



(a) Average Lifetime across all queries

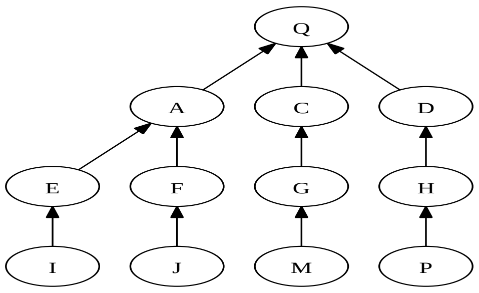


(b) Average Fidelity Loss across all queries

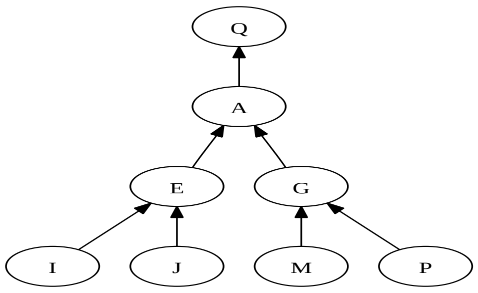
Fig. 11. Comparison of different path selection criteria against the number of queries.

4.2.6 *Comparison of our tree construction method with ad hoc tree construction method.* We performed experiments to compare the performance of our aggregation tree construction algorithm outlined in Section 3 with an ad hoc tree construction algorithm. In the ad hoc tree construction algorithm, a request injected at the query node is broadcast into the network. Each node that is a source for the aggregation query, responds with its node id. The responses are broadcast into the network till they reaches the query node. Each node chooses one of the neighboring nodes from which it receives the request as its dependent node. Figure 12 (a) shows a tree constructed using this ad hoc construction method. In Figure 12 (a) there is no possibility of in-network aggregation, whereas in Figure 12 (b) there is in-network aggregation at nodes E and G, close to the sources. With an AP-based scheme, for $c = 250$, the lifetime obtained with the ad hoc tree construction scheme is 2318 seconds whereas with our tree construction scheme it is 3132 seconds. Aggregation trees constructed using ad hoc tree construction schemes lead to higher fidelity loss (22%) as a result of higher message loss due to higher number of messages in the network. With our tree construction algorithm, aggregation trees obtained enable in-network aggregation, leading to a decrease in the number of messages in the network and thus a lower fidelity loss (of 6.66%).

4.2.7 *Comparison of prediction schemes.* Figure 13 shows a comparison of different prediction schemes presented in Section 2.4. The graph shows the *prediction inaccuracy* for the average of values sensed by 35 sensors for different values of coherency. Prediction inaccuracy is defined as the percentage of times the value of partial aggregate computed by a node differs from the value estimated by the



(a) Aggregation tree constructed using adhoc tree construction algorithm



(b) Aggregation tree constructed using our tree construction algorithm

Fig. 12. Comparison of trees constructed using different tree construction algorithms

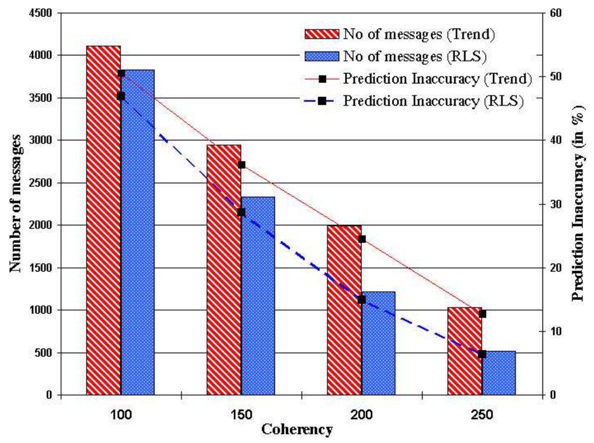
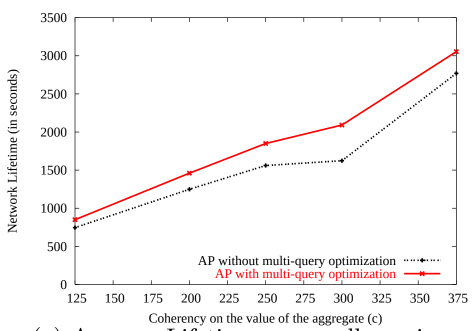
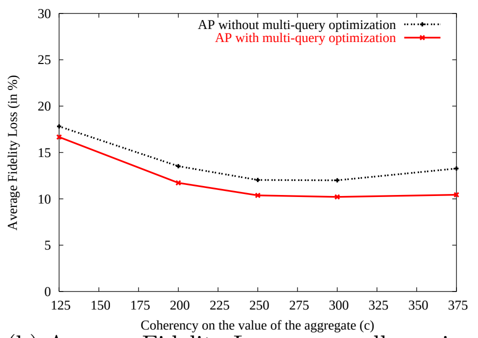


Fig. 13. Comparison of trend-based and RLS-based prediction schemes



(a) Average Lifetime across all queries



(b) Average Fidelity Loss across all queries

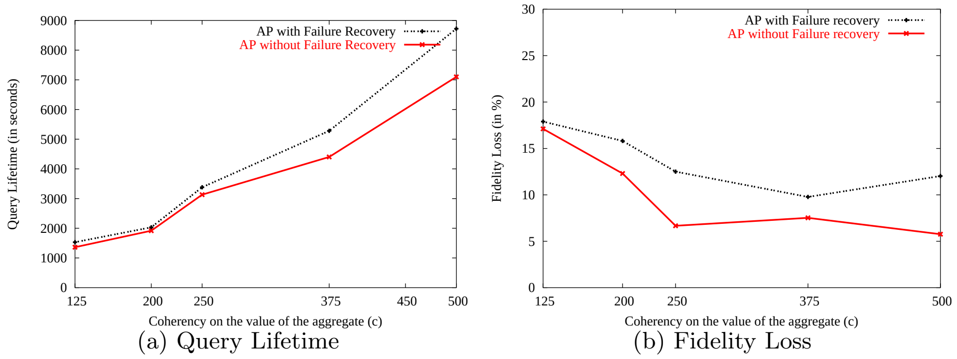
Fig. 14. Performance of Multi-query Optimization

dependent by more than $\alpha \times c'$. Over all values of coherency, on an average the RLS-based prediction scheme gives 8% higher prediction accuracy in comparison to the trend-based prediction scheme. The improvement in prediction accuracy leads to a decrease in the number of messages injected into the network by the sensors. Figure 13 also shows the comparison of the number of messages injected into the network by the partial aggregator node that computes the average of the values sensed by the 35 sensors. From the graph it can be said that for coherency $>= 150$, the node injects 20% lower number of messages into the network with the RLS-based scheme. We conclude that using RLS-based prediction instead of trend-based prediction in AP will lead to higher query lifetimes due to increased prediction accuracy. Given the improved accuracy of RLS, the already superior fidelity and lifetime delivered by AP will only get better if RLS were used instead of data trend for prediction. Our choice of using trend-based prediction method with AP in our experiments is due to its ease of implementation and lesser computational overhead compared to the RLS-based prediction method.

4.2.8 *Results for multiple aggregate queries.* We now present results for the performance of our optimization for multiple aggregate queries. Experiments were run with 4 aggregate queries each requesting an average over values sensed by 16 nodes. Each query had a target region containing 10 nodes in common with one other query. Figure 14 shows the performance comparison between AP with MQO and without MQO for different values of coherency requirements.

From the graphs, it can be seen that MQO gives an average of 20% improvement in lifetime over all values of coherency requirements. This increase is due to the decreased number of transmissions obtained by sharing common subtrees that compute the partial aggregate of nodes in the target region. AP with MQO gives an average of 15% decrease in loss of fidelity over the setup without MQO. The number of messages injected into the network is higher without MQO. This leads to an increase in number of AGGREGATE messages lost due to collisions in the network, thus leading to lower fidelities.

4.2.9 *Recovery from permanent failures.* Sensor nodes fail permanently either due to energy exhaustion or due to other external factors. We experimented with the scenario where sensors fail only due to energy exhaustion. Figure 15 shows the variation in lifetime and fidelity loss for our failure recovery mechanism. Experiments were run with different number of sources for an aggregate query with different coherency requirements. The failure of the nodes in these experiments is only due to energy exhaustion. A node broadcasts a DEATH message if its energy falls below 150 units. Note that 150 units is the amount of energy that is sufficient to perform one message send and one message receive. The recovery scheme gives an average lifetime improvement of about 20% along with fidelities of about 85%, lower than those obtained without failure recovery. The increase in lifetime of the query is due to local path repair done by our recovery mechanism, thus leading to availability of data to the query node for a longer amount of time. Also, during the path reconstruction process, updates to the aggregate are not received by the query node, thus leading to higher drop in fidelity when failure recovery is used. We expect a similar behavior when sensors fail due to external factors.



(a) Query Lifetime (b) Fidelity Loss

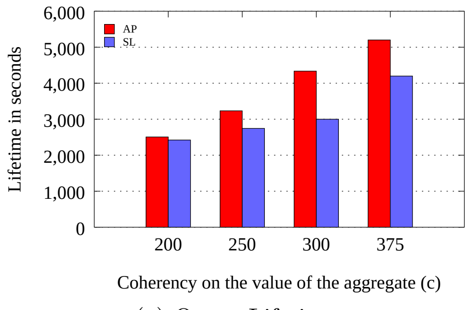
Fig. 15. Recovery from failures

4.2.10 *Experiments with intermittent node failures.* In addition to permanent failures, in a real world sensor network deployment, there arise cases were nodes may go down for a brief period of time, and come back up again. For example, a node might temporarily be disconnected from the rest of the network due to occlusion by a moving object in the network, or interference. We experiment with such intermittent failures by alternating between failed state and normal operation of a sensor. Owing to the nature of these failures, the duration for which a sensor is in the failed state is typically smaller than the duration of its normal operation. To account for this, the time (in seconds) for the failed state of the node is chosen by sampling from a uniform $[0, 2.5]$ distribution and that for the normal operation is sampled from a uniform $[0, 20]$ distribution.

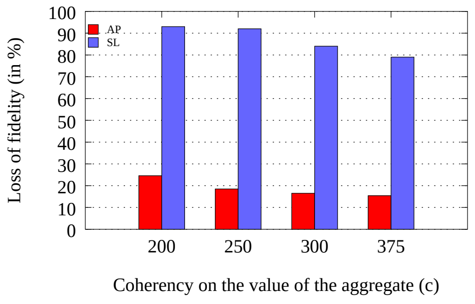
From the results in Figure 16, we observe that AP outperforms SL, giving fidelity losses below 20% while SL gives fidelity losses in the range of $80\% - 90\%$. This can be attributed to the predictive ability of AP to generate reasonably accurate values for partial aggregates and sources which are in the failed state. AP also yields higher query lifetimes, the improvement being approximately 25%. Improvement in lifetime is significant for higher values of user specified coherency, illustrating the need for lesser message transmissions to correct the prediction model as a result of the intermittent node failures.

4.2.11 *Effect of message losses.* In this section, we study the effect of message losses on AP. Figure 17 shows the results of our experiments with $c = 250$. As message losses increase, the lifetime of the query increases. A successful message transmitted to a dependent triggers a chain of updates along the path towards the query node. Higher loss rates lead to decrease in these events, leading to increased lifetime and higher loss in fidelities. Even with message losses as high as 15% AP outperforms SL.

4.2.12 *Results for complex aggregate (*`group by`*) queries.* In this section we present results for our experiments with `group by` queries. A network of 300 nodes was used for these tests. A query with the target region containing 4 groups was injected into the network. The target region for each group contains 25 nodes.

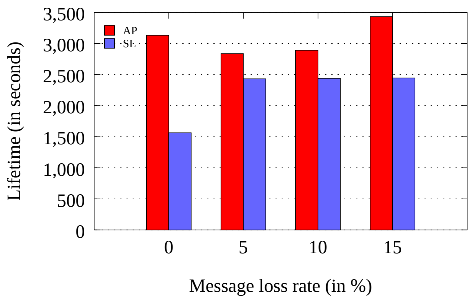


(a) Query Lifetime

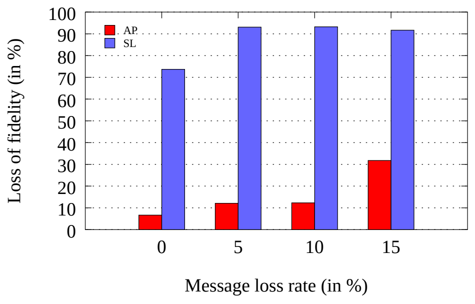


(b) Fidelity loss

Fig. 16. Effect of intermittent node failures on AP and SL



(a) Query Lifetime

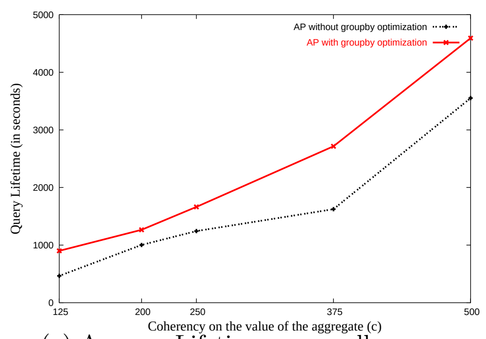


(b) Fidelity loss

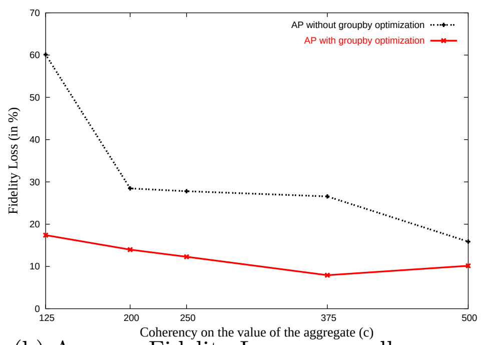
Fig. 17. Effect of message loss on AP and SL

We define the lifetime of a group as the amount of time for which the query node gets results from a majority of nodes belonging to that group. Figure 18 shows the performance of AP with and without the optimization for `group by` queries. The graphs show the variation of average of the lifetime and fidelity loss across all groups against user-specified coherency on the value of the aggregate. From the figure, the optimization for `group by` queries leads to an average increase of 40% in terms of lifetime and a 60% decrease in loss of fidelity. The increased lifetime may be attributed to the higher lifetimes obtained by routing partial aggregates for different groups possibly via different dependents.

4.2.13 *Prediction for other aggregates.* We evaluate the inaccuracy of prediction for three commonly used aggregates – average, maximum and minimum. From figure 19, we observe that the prediction inaccuracy is comparable for all the three aggregates for both the prediction schemes described in Section 2.4. For a user specified coherency of 200, the prediction inaccuracies for average, max, min respectively are 24.5%, 24.1% and 24.35% with the trend-based scheme and 14.88%, 15.27% and 15.2% with the RLS-based scheme. Prediction inaccuracies for other values of coherency requirements follow a similar trend. We thus conclude that in-network prediction can be used for computation of any of these commonly used aggregates.



(a) Average Lifetime across all groups



(b) Average Fidelity Loss across all groups

Fig. 18. Performance of optimization for `group by` queries

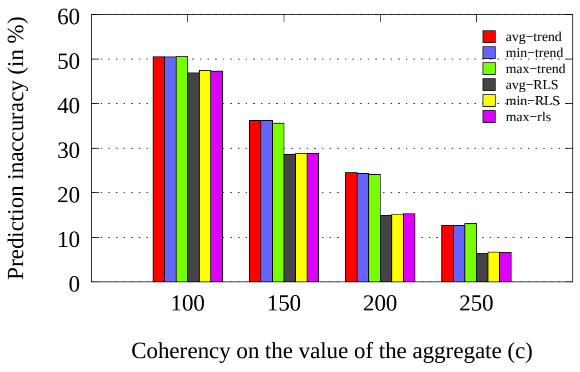


Fig. 19. Variation of prediction inaccuracy with coherency for min, max and average. σ denotes the standard deviation of the data trace.

5. RELATED WORK

Our first contribution is asynchronous computation of partial aggregates. Approaches like TAG [Madden et al. 2002], Cougar [Yao and Gehrke 2002] and TiNA [Sharaf et al. 2003] use synchronous computation of aggregates in-network. While synchronization provides a way to perform in-network aggregation which leads to reduction in the number of messages transmitted, it leads to loss in fidelity. Our approach recognizes the inherent asynchronism in the sensing activity of nodes and uses an asynchronous method for computing aggregates, thus delivering higher fidelity.

Our second contribution is in-network prediction of partial aggregates. [Madden et al. 2002; Yao and Gehrke 2002; Sharaf et al. 2003; Olston et al. 2001] use the last received value from a serving node to compute partial aggregates. Our approach predicts the value of partial aggregate at the serving nodes, thus providing resilience to loss of partial aggregate messages. Prediction in sensor networks finds mention in [Woo et al. 2003] which uses estimation to gather adaptive link connectivity statistics and thus make efficient routing decisions. Prediction based on sensed values has been explored in [Goel and Imielinski 2001; Papadimitriou et al. 2003; Li et al. 2006]. In [Li et al. 2006], the model uses proxies that are energy-rich and computationally unconstrained. The sensor nodes in the network do not create any prediction models, instead they use models supplied by the proxies. In contrast, all sensors in our setup, compute model parameters. [Deshpande et al. 2004] looks at a way to answer tolerance based queries in sensor networks by building a probabilistic data model based on multi-variate gaussians. When a new value is received, the model is updated by recomputing a new pdf by conditioning on the observed value. However, these techniques rely on the base station to build a common static prediction model for all sources. This does not leverage the benefits of in-network aggregation. In contrast, we propose in-network prediction of aggregates by in-network aggregator nodes on the aggregation tree using computationally inexpensive methods.

[Chu et al. 2006] propose an approach called *Ken* which uses replicated dynamic probabilistic models to answer SELECT * kind of queries in sensor networks. This work leverages spatio-temporal correlation in sensor readings to keep the probabilistic models at the query node up-to-date with those at the sources. Their use of clusters (referred to as *cliques*), to account for spatial-correlation of sensed data, may be detrimental to lifetime of the query due to possibility of failure of the cluster head. Our approach achieves both distribution of load in the network and recovery from node failures. [Cormode et al. 2005] propose a method to compute approximate quantiles over data generated by physically-distributed streams. They present a rate-based prediction model using average of historic values for tracking data dynamics. They extend this approach to a hierarchical structure (like the aggregation tree in our case) to present its applicability to sensor network streams. Both [Chu et al. 2006] and [Cormode et al. 2005] only try to reduce the error in data seen at the query node over the data generated at the sources, without attempting to provide temporally coherent data. In AP, we have used asynchrony as a means to obtain high fidelity of 90% or more.

In-network filtering has been used in the context of disseminating dynamic web data [Shah et al. 2003]. In AP, we exploit coherency requirements for filtering out updates to minimize energy consumption. So does TiNA [Sharaf et al. 2003]. In [Olston et al. 2003], an approach for reducing communication overhead by adjusting filters set over individual data streams in a environment of distributed streams is presented. Filters on the sources are dynamically modified so that each query in a multi-query setup receive answers at the user-specified precision.

[Madden et al. 2002; Yao and Gehrke 2002] propose in-network aggregation on aggregation trees. However, the ad hoc routing algorithms they use for tree construction may often construct trees where the performance gain that can be provided by in-network aggregation is not entirely leveraged. Our experiments revealed that ad hoc tree construction methods often lead to creation of aggregation trees (such as those shown in Figure 12 (a)) where in-network aggregation is not possible since the paths from different sources to the query node intersect only at the query node. In contrast, we consider the remaining lifetime of sensors and delays along dissemination paths and try to maximize the energy savings offered by in-network aggregation by aggregating it as close to the data sources as possible. Tributaries-Deltas [Manjhi et al. 2005] combines the tree-based and multipath-based routing approaches by running them in different parts of the network. Nodes switch between running tree-based and multipath-based routing algorithms depending on current message loss rates. Synopsis diffusion [Nath et al. 2004] uses a multi-path based approach to route updates (encoded as synopses) from the data sources to the query node. The same message may be received via two or more different paths. Computing approximations to the value of the required aggregate entails the overhead of avoiding double-counting for duplicate sensitive aggregates.

Finally, in this paper we propose a solution to the problem of multi-query optimization. Previous approaches to solve this problem [Trigoni et al. 2005] do not exploit energy optimizations that can be obtained by creating dissemination trees which share subtrees. Dissemination tree for all the source sensors in a set of queries is created, and possible candidates for subtree sharing are identified afterwards. In contrast our approach creates trees that identifies dissemination subtrees during the tree construction phase.

6. SUMMARY AND FUTURE WORK

In this paper, we studied various issues involved in answering aggregate queries in sensor networks with focus on result quality and query lifetime. We proposed an efficient solution called Asynchronous in-network Prediction for answering aggregate queries. Experimental results demonstrate that AP gives higher query lifetimes and result quality than synchronous last-valued-based aggregation methods. AP recognizes the inherent asynchronism in the sensing activity of nodes and uses an asynchronous method for computing aggregates, thus delivering higher fidelity. In constructing the aggregation tree for routing data, we make an attempt to maximize the benefits of in-network aggregation by building an aggregation tree that performs in-network aggregation as close to the sources as possible. Further, we presented a novel techniques for optimizing across multiple aggregate queries and for complex aggregate queries with `group by` clauses. As part of ongoing work, we also plan to test our algorithms on a sensor testbed.

REFERENCES

Chu, D., Deshpande, A., Hellerstein, J. M., and Hong, W. 2006. Approximate data collection in sensor networks using probabilistic models. In *ICDE*.

Cormode, G., Garofalakis, M., Muthukrishnan, S., and Rastogi, R. 2005. Holistic aggregates in a networked world: distributed tracking of approximate quantiles. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 25–36.

Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J. M., and Hong, W. 2004. Model-driven data acquisition in sensor networks. In *VLDB*. 588–599.

Goel, S. and Imielinski, T. 2001. Prediction-based monitoring in sensor networks: Taking lessons from MPEG. *SIGCOMM Comput. Commun. Rev. 31,* 5.

Hill, J. and Mica, D. C. 2002. A wireless platform for deeply embedded networks. *IEEE Micro. 22(6).*

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. 2000. System architecture directions for network sensors. *ASPLOS*.

Intanagonwiwat, C., Govindan, R., and Estrin, D. 2000. Directed Diffusion: A Scalable and

Robust Communication Paradigm for Sensor Networks. *Sixth International Conference on Mobile Computing and Networking*.

Inventory of U.S. Globec Georges Bank Data, Project: AL9508. 1995. http://jgof.wh.whoi.edu/jg/serv/globec/gb/brdscale/al_shipdata.html1%7Bdir=globec.whoi.edu/jg/dir/globec/gb/broadscale/,info=globec.whoi.edu/jg/info/globec/gb/broadscale/alongtrack%7D?cruise_id%20%eq%20al9508.

Kaiser, W. J. WINS NG 1.0 Transceiver Power Dissipation Specifications. *Sensoria Corp*.

Levis, P., Lee, N., Welsh, M., and Culler, D. 2003. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *First ACM Conference on Embedded Networked Sensor Systems(SenSys)*.

Li, M., Ganesan, D., and Shenoy, P. 2006. PRESTO: Feedbackdriven Data Management in Sensor Networks. In *In ACM/USENIX Symposium on Networked Systems Design and Implementation*.

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2002. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev. 36,* SI.

Manjhi, A., Nath, S., and Gibbons, P. B. 2005. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*.

Nath, S., Gibbons, P. B., Seshan, S., and Anderson, Z. R. 2004. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, New York, NY, USA, 250–262.

Olston, C., Jiang, J., and Widom, J. 2003. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM Press, New York, NY, USA, 563–574.

Olston, C., Loo, B. T., and Widom, J. 2001. Adaptive precision setting for cached approximate values. *SIGMOD Rec. 30,* 2, 355–366.

Papadimitriou, S., Brockwell, A., and Faloutsos, C. 2003. Adaptive, hands-off stream mining. In *VLDB*.

Shah, S., Dharmarajan, S., and Ramamritham, K. 2003. An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data. *29th Very Large Data Bases Conference*.

Sharaf, M. A., Beaver, J., Labrinidis, A., and Chrysanthis, P. K. 2003. TiNA: A Scheme for Temporal Coherency-Aware in-Network Aggregation. *Third International ACM Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*.

TinyOS Website. http://www.tinyos.net.

Trigoni, N., Yao, Y., Demers, A. J., Gehrke, J., and Rajaraman, R. 2005. Multi-query optimization for sensor networks. In *DCOSS*. 307–321.

Woo, A., Tong, T., and Culler, D. 2003. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*.

Yao, Y. and Gehrke, J. 2002. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31,* 3, 9–18.

Young, P. 1984. *Recursive estimation and time-series analysis: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA.