Aggregate Threshold Queries in Sensor Networks

Izchak Sharfman[1], Assaf Schuster[1], Daniel Keren[2]

[1]Technion
Dept. of Computer Science
Hafia, Israel

[2]Haifa University
Dept. of Computer Science
Haifa Israel

Abstract

An important class of queries over sensor networks are network-wide aggregation queries. In this work we study a class of aggregation queries which we refer to as aggregate threshold queries. *The goal of an aggregate threshold query is to continuously monitor the network and give a notification every time an aggregated value crosses a predetermined threshold value. Aggregate threshold queries are of particular importance in a wireless sensor environment, since they allow network-wide events to be detected, with a minimum expenditure of energy. Such network-wide events might include, for example, the variance in sensor readings exceeding a certain threshold.*

We present an efficient algorithm for implementing arbitrary aggregate threshold queries over sensor networks. Our algorithm is based on a novel geometric approach by which an arbitrary aggregate threshold query can be split into a set of numerical constraints on the readings of the individual sensors. These constraints are used by the individual sensors to monitor their readings. The constraints are constructed so that as long as none of the constraints are violated, it is guaranteed that the aggregated value has not crossed the threshold. Experiments we performed on real-world data indicate that by employing these constraints, sensors are able to reduce the number of transmissions required for implementing the query by orders of magnitude, thus significantly reducing energy consumption.

1 Introduction

Wireless sensor networks are a powerful tool for performing high level monitoring tasks. Sensor networks have been suggested for a wide variety of applications, including military applications (detecting and tracking vehicles behind enemy lines), ecological applications (monitoring an ecological system), civil engineering applications, and disaster control applications (monitoring the advance of a hazardous gas cloud).

Many deployments of sensor networks are tiered, i.e., the network consists of two types of units: many inexpensive wireless sensing devices, referred to as motes, and fewer, more powerful control units, referred to as macro-nodes [6, 10, 17]. Motes are simple wireless units, typically based on simple 8-bit processors, and are very limited in their processing, memory, and energy resources. Macro-nodes, on the other hand, are more resource rich in terms of processing power, memory, and wireless communications. They are typically based on more advanced 32-bit processors, enabling them to run advanced operating systems. Furthermore, macro-nodes are much less energy constrained.

The network is divided into clusters, each cluster consisting of a group of motes and a single macro-node. The motes in a cluster are referred to as cluster members, while the macro-node is referred to as the cluster head. The role of motes is limited to sensing the environment based on instructions received from their cluster head, and reporting readings to their cluster head. In addition, a mote may be involved in the multi-hop routing of data from another mote in the cluster to the cluster head. Macro-nodes are responsible for performing the application level functionality of the monitoring task at hand. Macro-nodes use their superior communications capabilities to implement a network wide communications backbone. Routing between macro-nodes is performed solely by the macro-nodes, i.e., the motes do not participate in message passing between the macro-nodes.

This tiered architecture has several advantages. First, recent work indicates that this tiered model has advantages in terms of performance over single tiered architectures [3, 8, 16, 18]. In addition, since application logic is run on stronger devices that support more advanced development tools and operating systems, the tiered model significantly shortens development and deployment cycles [3]. Finally, since most of the units are inexpensive motes (usually the

1-4244-0910-1/07/$20.00 ©2007 IEEE.

number of motes is one or two orders of magnitude greater than the number of macro-nodes), deploying a tiered network remains relatively inexpensive.

Typically, the purpose of a sensor network is to provide global insights regarding the state of the monitored environment, as opposed to providing a set of raw data readings. An important method for obtaining global insights is calculating global aggregates over the sensors: for example a sensor network monitoring an ecological system may be queried in order to determine the average temperature in the monitored environment. Aggregates also play an important role in the maintenance of the sensor network itself. For example, the network may be queried in order to determine the average residual energy level of a mote in the network.

In recent years several schemes have been proposed for aggregating data in a sensor network [5, 9]. While these schemes enable the efficient computation of aggregates, they are "one-shot" schemes, i.e., the aggregate is calculated using the data held by the sensors at a specific point in time. The algorithm proposed in [19] allows continuous queries over sensor networks, i.e., the algorithm constantly updates the calculated aggregate to reflect the aggregate of the current values held by the sensors, while minimizing the amount of data transmitted by the sensors. While continuously providing an updated global aggregate can be useful in some cases, in other cases we may require the network only to report alert conditions, characterized by a global aggregate crossing a threshold value. We refer to this type of continuous query as an *aggregate threshold query*.

Consider, for example, a tiered sensor network, deployed in a large server room or conference room, where each mote is equipped with a temperature meter. The purpose of the sensor network is to monitor the temperature in the room and control the air conditioning system to maintain constant and uniform temperature. In order to regulate the amount of cold air produced by the air conditioning system, we would like the sensor network to alert us when the average temperature in the room exceeds a maximum value or drops below a minimum value. We also would like to eliminate any "hot spots" that may be caused by a rack of active servers in a server room or an intensely lit stage in a conference room. In order to ensure uniform temperature throughout the room, we would like the sensor network to alert us when the variance in the temperature readings exceeds a certain threshold value. Upon receiving a variance alert, the system will gather detailed readings from the sensors and redistribute the flow of air from the system to eliminate these "hot spots".

Note that detecting a high variance in the temperature is much more difficult than detecting a high average temperature. When the average temperature reading taken by the motes exceeds a certain threshold, the temperature reading taken by at least one of the motes in the room is necessarily above the threshold, therefore a high average temperature always has a local indication. On the other hand, a high variance in the temperature may not have a local indication. Consider, for example, a network comprising of 5 evenly sized clusters. The cluster head of the i^{th} cluster can communicate with the cluster head of clusters $i-1$ and $i+1$. Say that all the motes in a given cluster read the same temperature, but the temperature read by motes in the first cluster is 20 degrees, and the temperature read by motes in the fifth cluster is 24 degrees. Furthermore, say that the difference in the readings of two adjacent clusters is 1 degree. Say we are interested in detecting when the variance in the temperature readings exceeds 1.8 deg^2. The variance inside each cluster is 0 (all the motes read the same temperature). The variance in the temperature read by motes in three consecutive clusters is $\frac{2}{3}$ deg^2. But the variance in the temperature in the entire network is 2 deg^2. This example demonstrates that in some cases, a variance in the readings of motes may not have a local indication, or in other words, detecting that the variance in the readings of motes exceeds a given threshold may require collecting data from the entire network.

More formally, a continuous aggregate threshold query can is defined as follows: we assume that every mote takes a set of measurements represented as a vector of real values. Let the aggregation function be an arbitrary function taking a measurement vector and returning a real value. We are interested in determining at any given time whether the value of the aggregation function on the *average* of the measurement vectors taken by the individual motes has crossed a predetermined threshold.

In many cases determining whether an aggregate has crossed a threshold does not require knowing the exact value of the aggregate, therefore, providing an alert when an aggregate (the average or variance in temperature in the example given above) crosses a threshold should require less communication than continuously providing an estimate of the aggregate. Current solutions for monitoring aggregates in sensor networks do not exploit this fact in order to reduce communication for thresholded aggregate queries.

The main contribution of this paper is a communication efficient algorithm (and therefore it is also a power efficient algorithm) for performing aggregate threshold queries. The algorithm is based on a method for decomposing the query into a set of constraints on the values held by each mote. As long as none of these constraints has been violated, no communication is required.

2 Related Work

Aggregation queries over sensor networks have received considerable attention in recent years [2, 4, 5, 7, 8, 9, 11, 12, 15, 19]. Madden et al. [9] propose a tree based algorithm for performing aggregation queries. The algorithm consists of two phases: in the first phase, the query is propagated down a spanning tree which is constructed over the network. Once the query has reached the leaves of the spanning tree, they send their readings to their parents. Once a sensor has received readings from all its children, it aggregates its local reading with the ones sent by its children, and sends the aggregated value to its parent. Greenwald et al. [4] present an algorithm for calculating quantiles where in a network with *n* sensors, each sensor transmits $\mathrm{O}(\log^3 n)$ bits. Shrivastava et al. [15] present an algorithm for approximating quantiles over sensor networks where each sensor transmit a fixed number of bits. Tree-based algorithms are "one-shot", i.e. calculate an aggregate on a "snapshot" of the data.

An alternative to the tree approach is the multi-path approach [2, 12]. In the multi-path approach, the sensors are partitioned into a set of rings. The i^{th} ring consists of all the sensors that are i hops from the base station. Aggregation is performed from the most distant ring, towards the base station. In contrast to the tree approach where each sensor sends its intermediate aggregate to a designated parent sensor, in multi-path approach each sensor broadcasts its intermediate aggregate, which is processed by all the sensors in the subsequent ring that have received the broadcast. Intermediate aggregates are represented by special sketch structures, that are resilient to double counting of values that may occur due to an intermediate aggregate being processed by multiple sensors in the subsequent ring. The advantage of the multi-path approach is that it is more resilient to packet loss, but on the other hand, since the aggregation is performed on sketches of the values rather than on the values themselves, the multi-path approach provides approximated aggregates, whereas tree-based approaches can provide accurate aggregates in case data delivery is reliable.

Manjhi et al. [11] proposed an algorithm that combines the advantages of both approaches. The algorithm uses the tree-based approach for aggregating data from sensors that are far from the base station, a multi-path approach for aggregating data from sensors that are close to the base station, and dynamically determines where to switch from the tree-based approach to the multi-path approach according to network conditions. As tree-based algorithms, multi-path algorithms are "one-shot" algorithms, whereas our algorithm is specifically designed to handle continuous queries.

Zhao et al. [19] present an algorithm for continuously evaluating simple aggregated values, such as sums, counts and averages, over a sensor network. Our work also deals with continuous queries, but it differs from their work in two respects. First, the algorithm presented in [19] continuously provides an estimate of the aggregated value, whereas our algorithm handles thresholded aggregate queries. Second, while the work in [19] focuses on simple aggregates, our algorithm supports aggregates that can be expressed by arbitrary functions.

Our algorithm is designed for a tiered sensor network. While most algorithms for sensor network consider a single tier network (i.e. all the sensors in the network are identical), in practice most real world deployments of sensor networks are tiered. Examples include [6, 10, 17]. Previous studies of tiered sensor networks include [16, 3, 8, 14, 18].

Finally, algorithms for monitoring arbitrary threshold functions over distributed streams have been presented in [13]. [13] presents two monitoring algorithms, for two different computational environments. The first algorithm, referred to as the decentralized algorithm, assumes that all nodes share a broadcast domain, i.e. any message sent by one of the nodes is received by all other nodes. The second algorithm, referred to as the coordinator based algorithm, designates one of the nodes as a coordinator, and assumes that all the nodes communicate solely with the coordinator. Both algorithms are based on a geometric interpretation of the monitoring problem.

Neither of the algorithms presented in [13] are suitable for sensor networks, since implementing the primitives they assume in a sensor network (either a single broadcast domain or global communications with a single node) is very costly in terms of energy expenditure. This work adopts the geometric interpretation presented in [13], but proposes an algorithm that is able to resolve conflicts locally, and is therefore sensitive to energy expenditure constraints.

3 Computational Model

We denote the number of clusters in the network by *n*. We denote the number of motes in the i^{th} cluster by N_i, and the total number of motes in the network by N_{tot}. We denote the motes in the i^{th} cluster by $s_{i,1}, s_{i,2}, ..., s_{i,N_i}$, and the cluster head of the i^{th} cluster by m_i. We assume that all the motes in a cluster (or at least the majority of them) have a direct radio link with the cluster head, and that all radio links are full duplex. The motes in a cluster only interact with their cluster head.

Apart from interacting with the motes in their cluster, cluster heads also communicate with the cluster heads of neighboring clusters. The links between the cluster heads are modeled by an undirected connectivity graph, where each cluster is represented by a vertex, and an edge connects every two vertices that represent neighbouring clusters. We assume that a spanning tree has been constructed over the connectivity graph, as in [9, 19]. We assume that a query is injected into the network from one of the cluster heads, referred to as the *gateway node*, and denoted by m_1.

A continuous aggregate threshold query is formally defined as follows: each mote $s_{i,j}$ holds a *d*-dimensional vector of measurements, which is referred to as the *measurement vector* and is denoted by $\vec{v}_{i,j} = (v_{i,j}^{(1)}, v_{i,j}^{(2)}, ..., v_{i,j}^{(d)})^T$. Let the *global measurement vector*, denoted by $\vec{v} =$

$(v^{(1)}, v^{(2)}, ..., v^{(d)})^T$, be the average of the measurement vectors held by all the motes in the network:

$$\vec{v} = \frac{1}{N_{tot}} \sum_{i=1}^{n} \sum_{j=1}^{N_i} \vec{v}_{i,j}$$

Let a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$, called the *aggregation function*, be an arbitrary function from the space of the *d*-dimensional real vectors to the reals. Let *r* be a predetermined threshold. A continuous aggregate threshold query requires that any sensor will be able to determine at any given time whether or not $f(\vec{v}) > r$.

As an example, the average temperature alert described in Section 1 can be trivially formulated as an aggregate threshold query: the measurement vector held by each mote is a scalar, holding the temperature reading at the mote, and the aggregation function is the identity function, $f(x) = x$. The temperature variance alert can be formulated as an aggregate threshold query, as follows: let X denote a random variable representing a set consisting of the temperature reading of the individual motes. Let $x_{i,j}$ denote the temperature reading at the mote $s_{i.,j}$. Each mote holds the following measurement vector:

$$\vec{v_{i,j}} = \left(\begin{array}{c} x_{i,j} \\ (x_{i,j})^2 \end{array} \right)$$

Note that

$$\vec{v} = \frac{1}{N_{tot}} \sum_{i=1}^{n} \sum_{j=1}^{N_i} \left(\begin{array}{c} x_{i,j} \\ (x_{i,j})^2 \end{array} \right) = \left(\begin{array}{c} E[X] \\ E[X^2] \end{array} \right)$$

Therefore, the variance of the temperature readings can be calculated using the following aggregation function:

$$f(\vec{v}) = v^{(2)} - (v^{(1)})^2 = E[X^2] - E[X]^2 = Var(X)$$

4 A Network with a Single Cluster

In this section we describe an algorithm implementing aggregate threshold queries in a network consisting of a single cluster. In Section 5 we extend this algorithm to handle networks consisting of multiple clusters. Recall from Section 3 that each mote s_i holds a measurement vector denoted by $\vec{v}_i$ (since in this section we limit our discussion to a network consisting of a single cluster, we omit the first index, i.e., the cluster index, from mote and measurement references).

From time to time, as dictated by the algorithm, the cluster head collects the current measurements vectors from all the motes, calculates their average, and reports the average vector to all the motes. This action is referred to as *synchronizing* the motes.

The average vector calculated by the cluster head during the last synchronization event is called the *estimate vector*, and is denoted by $\vec{e}$. Each mote remembers the measurement vector collected from it by the cluster head during the last synchronization event. The measurement taken from the mote s_i during the last synchronization event is referred to as the *reference vector*, and is denoted by $\vec{v}_i{}'$. According to these definitions:

$$\vec{e} = \frac{1}{N} \sum\nolimits_{i=1}^{N} \vec{v}_i{}'$$

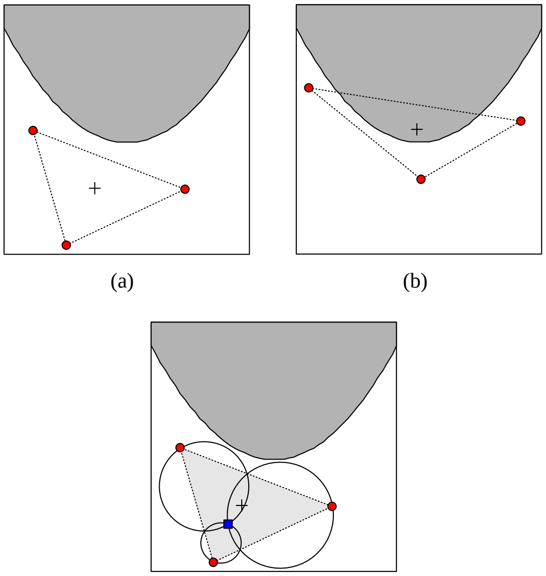
Finally, each mote maintains two additional variables. The first variable is called the *slack vector*, and is denoted by $\vec{\delta}_i$. The algorithm will guarantee that at any given time the sum of the slack vectors held by the motes is 0 $(\sum\limits_{i=1}^{n} \vec{\delta}_i = 0)$. In particular, after a synchronization event, each mote sets its slack vector to 0. The second variable maintained by each mote is called the *drift vector*, denoted by u_i. The drift vector is calculated as follows:

$$\vec{u}_i = \vec{e} + \vec{v}_i - \vec{v}_i{}' + \vec{\delta}_i \qquad (1)$$

It is easy to show that at any given time, the average of the drift vectors held by the motes is equal to the global measurement vector. Furthermore, immediately after the motes are synchronized, for every mote, s_i, the reference vector is equal to the measurement vector ($\vec{v}_i{}' = \vec{v}_i$) and the slack vector is 0, therefore the drift vector held by each mote immediately after a synchronization event is equal to the estimate vector.

The algorithm is based on the following geometric interpretation of an aggregate threshold query: vectors held by motes are viewed as points in $\mathbb{R}^d$. The combination of the aggregation function, *f*, and the threshold value, *r*, defines the following coloring over $\mathbb{R}^d$: any point $\vec{x} \in \mathbb{R}^d$ for which $f(\vec{x}) < r$ is said to be white, while any point $\vec{y} \in \mathbb{R}^d$ for which $f(\vec{y}) \geq r$ is said to be gray. Figure 1 depicts the coloring induced by the VARIANCE aggregate function, $f(\vec{x}) = x^{(2)} - (x^{(1)})^2$ ($x^{(1)}$ is plotted on the horizontal axis, and $x^{(2)}$ is plotted on the vertical axis), and a threshold *r*=1.8 deg^2. The goal of the aggregate threshold query under this interpretation is to determine the color of the point representing the global measurement vector. Figure 1 depicts the drift vectors held by three motes (the red circles), and the global measurement vector they define, i.e., the average of these vectors (the cross). In both Figure 1(a) and Figure 1(b), the drift vectors held by the motes are white, but in 1(a) the global measurement vector is white, while it is gray in 1(b). This demonstrates that the query cannot be answered solely by observing the color of the drift vectors held by the motes.

Every time the measurements taken by a mote change, the mote checks that the new measurement vector complies



(a)

(b)

(c)

Figure 1. Geometric interpretation.

to a local constraint. These constraints are constructs such that if the constraints on all motes are satisfied, it is guaranteed that the convex hull of the drift vectors is monochromatic (i.e. all the vectors belonging to the convex hull have the same color). Since the global measurement vector is the average of the drift vectors held by the motes, it belongs to the convex hull of the drift vectors. Therefore, if the convex hull is monochromatic, the color of the global measurement vector is the same as the color of the convex hull.

At first glance it may seem difficult to determine if the convex hull of the drift vectors is monochromatic by setting local constraints on the individual drift vectors, since as demonstrated in Figure 1(b), only knowing that the drift vectors are monochromatic is insufficient for determining that their convex hull is monochromatic. In order to verify that the convex hull of the drift vectors is monochromatic, we use Theorem 4.1, taken from [13].

Theorem 4.1. *Let* $\vec{x}, \vec{y}_1, \vec{y}_2, ..., \vec{y}_n \in \mathbb{R}^d$ *be a set of vectors in* $\mathbb{R}^d$. *Let Conv*$(\vec{x}, \vec{y}_1, \vec{y}_2, ..., \vec{y}_n)$ *be the convex hull of* $\vec{x}, \vec{y}_1, \vec{y}_2, ..., \vec{y}_n$. *Let* $B(\vec{x}, \vec{y}_i)$ *be a ball centered at* $\frac{1}{2}(\vec{x} + \vec{y}_i)$ *and with a radius of* $\left\|\frac{1}{2}(\vec{x} - \vec{y}_i)\right\|_2$ *i.e.,* $B(\vec{x}, \vec{y}_i) = \left\{\vec{z} \,\middle|\, \left\|\vec{z} - \frac{1}{2}(\vec{x} + \vec{y}_i)\right\|_2 \leq \left\|\frac{1}{2}(\vec{x} - \vec{y}_i)\right\|_2\right\}$. *Then Conv*$(\vec{x}, \vec{y}_1, \vec{y}_2, ..., \vec{y}_n) \subset \bigcup_{i=1}^{n} B(\vec{x}, \vec{y}_i)$.

Theorem 4.1 is used to bound the convex hull of *n+1* vectors in $\mathbb{R}^d$ by the union of *n d*-dimensional balls. In our case it is used to bound the convex hull of the estimate vector and the drift vectors i.e., Conv$(\vec{e}, \vec{u}_1, \vec{u}_2, ..., \vec{u}_n)$, by a set of n balls, where each ball is constructed independently by one of the motes. Each mote, s_i, constructs a ball $B(\vec{e}, \vec{u}_i)$, which is centered at $\frac{\vec{e}+\vec{u}_i}{2}$, and has a radius of $\left\|\frac{\vec{e}-\vec{u}_i}{2}\right\|$. This ball is called the *drift sphere*. Note that at any given time each mote has all the information required to independently construct its drift sphere.

Theorem 4.1 states that the convex hull of the drift vectors and the estimate vector is bound by the union of the drift spheres constructed by the motes. Therefore, if all the drift spheres are monochromatic, the convex hull is guaranteed to be monochromatic, and thus the global measurement vector is the same color as the convex hull. Since the estimate vector is part of the convex hull as well, if all the drift spheres are monochromatic, the global measurement vector and the estimate vector have the same color. To test whether a ball is monochromatic, we calculate the maximum and minimum values of *f* over it. Due to lack of space we cannot elaborate on this any further; suffice to say that for important functions such as the variance, there is a very simple closed-form solution.

Figure 1(c) illustrates the use of Theorem 4.1. The setup depicted in Figure 1(a) is shown, together with the estimate vector (the blue square), and the drift sphere constructed by each of the motes. One can notice, that as stated by Theorem 4.1, the convex hull of the drift vectors is bound by the union of the drift spheres constructed by the motes.

In order to complete the description of the algorithm we need to specify how to resolve constraint violations. One method of resolving constraint violations is by synchronizing the motes. As mentioned earlier, synchronizing the motes produces a new estimate vector, and sets the drift vectors on all motes to be equal to the estimate vector. The new drift spheres held by the motes have 0 radius, and are therefore monochromatic by definition, and the constraints on all the motes are upheld. Synchronizing the motes is relatively costly because it requires communicating with all the motes. Therefore, a more efficient method called *balancing* is first used to attempt to resolve a constraint violation.

A balancing process attempts to resolve a constraint violation by constructing a set of motes, referred to as the *balancing group*, such that the balancing group includes the mote whose constraint has been violated, and the average of the drift vectors held by the motes in the group creates a monochromatic drift sphere. If a balancing group has been successfully constructed, all the motes in the group modify their drift vectors to be equal to the average of the drift vectors held by the motes in the balancing group by modifying their slack vectors. Note that the sum of the slack vectors in such a case remains 0, therefore the global measurement vector is guaranteed to remain in the convex hull of the drift vectors. The advantage of balancing is that it that as opposed to synchronization, it does not require all the nodes to process, and is therefore more efficient. If a constraint violation has not been resolved by a balancing process, the nodes are synchronized.

Balancing is performed as follows: first the mote whose constraint has been violated reports its drift vector to the cluster head. This mote is the first mote to be added to the balancing group, and is referred to as the *unbalanced mote*. The cluster head constructs the balancing group by iteratively adding new motes to the balancing group. In each iteration the cluster head randomly selects a number of motes that are not in the group, and requests them to send their drift vector. In the i^{th} iteration, the cluster head randomly selects 2^{i-1} new motes to be included in the balancing group. The average of the drift vectors held by the members of the balancing group is referred to as the *balanced vector*.

Note that since the cluster head communicates with the motes using broadcast messages, in each iteration the cluster head uses a single broadcast message to request drift vectors from all the selected motes. By doubling the number of motes that are added to the balancing group in each iteration, the number of messages the cluster head produces in order to complete the balancing process is logarithmic in the size of the cluster.

After each iteration the cluster head checks if the balanced vector create a monochromatic drift sphere. If so, the balancing process is said to have succeeded. If the balancing group includes all the motes in the cluster, and the balanced does not create a monochromatic drift sphere, the balancing process is said to have failed.

If the balancing process has succeeded, the cluster head sends the balanced vector to the members of the balancing group. Upon receipt of the balanced vector, each mote, including the unbalanced mote, sets its slack vector so that its drift vector is equal to the balanced vector, thus resolving the original constraint violation.

5 Multi-Cluster Networks

In this section we extend the algorithm presented in Section 4 to networks that comprise of multiple clusters. As described in Section 3, the network can be modeled by an undirected connectivity graph, where each cluster is represented by a vertex, and an edge connects every two vertices that represent neighbouring clusters. We assume that a spanning tree has been constructed over the graph.

The multi-cluster algorithm is also based on decomposing the query into a set of constraints, monitored locally by each mote. All the motes hold a common estimate vector. Each mote maintains a drift vector, and constructs a drift sphere according to the estimate vector and its drift vector. As long as the constraints on all the motes are upheld, no communication is required. In case a constraint is violated on a mote, an attempt is made to resolve it by balancing the violating drift vector. First an attempt is made to balance the violating drift vector with drift vectors held by members of the same cluster. This process is referred to as an *intra cluster balancing*. If unsuccessful, an attempt is made to balance the constraint violation with drift vectors held by members of other clusters. This process is referred to as *extra cluster balancing*. Finally, if the extra cluster balancing has failed, all the motes in the network are synchronized.

When a local constraint is violated on a mote, it sends its drift vector to its cluster head. The cluster head tries to resolve the constraint violation by performing an intra cluster balancing process, which is similar to the balancing process described in Section 4. If the cluster head failed to balance the violating vector, it will initiate an extra cluster balancing process.

Extra cluster balancing is performed by passing a token between cluster heads. The token contains two values, the *aggregation vector*, denoted by $\vec{a}$, which holds the average of the drift vectors held by the motes in the clusters the token has visited so far, and a counter, denoted by *c*, which holds the total number of motes in the clusters the token has visited so far. To initiate extra cluster balancing, the cluster head creates a token with an aggregation vector consisting of the average of the drift vectors held by the cluster members, and sets the token counter to the number of motes in the cluster. Note that the average drift vector has already been calculated by the intra balancing process, therefore no additional communication is needed in order to create the token. More formally, the cluster head m_i will create the following token:

$$< \vec{a}, c > \leftarrow < \frac{\sum_{j=1}^{N_i} \vec{u}_{i,j}}{N_i}, N_i >$$

If the drift sphere defined by a token's aggregation vector is monochromatic (i.e. $B(\vec{e}, \vec{a})$ is monochromatic) the token is said to be balanced, otherwise it is said to be unbalanced. The cluster head that created the token is referred to as the *source* of the token. When a cluster head, m_k, receives a token for the first time, it collects the drift vectors from its cluster members, and adds them to the average vector held by the token by modifying the token as follows:

$$< \vec{a}, c > \leftarrow < \frac{\sum_{j=1}^{N_k} \vec{u}_{k,j} + c \cdot \vec{a}}{N_k + c}, N_k + c > \qquad (2)$$

The token is passed between the cluster heads until the token has been balanced, or until it has visited all the clusters. If the token has been balanced, the extra cluster balancing process is said to have succeeded, and the motes in each of the clusters that have received the token will set their slack vector so that their drift vector will be equal to the aggregation vector held by the token. If the token has traversed all the clusters, and has not been balanced, the extra cluster balancing process is said to have failed, and all the motes in the network will adopt the aggregation vector held by the token as the new estimate vector (thus implementing a synchronization process). At this stage we assume that at any given time, only a single token traverses the network. Later we extend the algorithm to handle multiple tokens traversing the cluster heads.

In order to complete the description of the algorithm, we need to specify exactly how the token is passed among the cluster heads. Recall that we assumed that a spanning tree has been constructed over the connectivity graph. The token is passed over the edges of the spanning tree, according to a depth first search (DFS), rooted at the source of the token. This ensures that as long as the token is unbalanced, it will continue to traverse the clusters. If, during traversal of the token through the cluster heads, one of them detects that the token is balanced, the balanced token is flooded to all the cluster heads that the token traversed. Upon receipt of the token, each cluster head broadcasts the aggregation vector specified in the token to all the cluster members, which in turn set their slack vectors so that its drift vector is equal to the aggregation vector, thus implementing a successful balancing process. If the token has traversed all the clusters, and has not been balanced (this can be detected by the source of the token), the token is flooded to all the cluster heads in the network. Upon receipt of the token, each cluster head broadcasts the aggregation vector specified in the token to all the cluster members, which in turn set the estimate vector to be equal to the aggregation vector, and set their reference vector to be equal to the measurement vector that was collected from it by the cluster head, thus implementing a synchronization process.

5.1 Handling Multiple Tokens

When presenting the multi-cluster algorithm, we assumed that at any given time only a single token traverses the network. In practice, several constraint violations may occur simultaneously on motes in several different clusters. This may lead to several tokens simultaneously traversing the network. Therefore, the multi-cluster algorithm must be extended to specify how to handle cases where a token has reached a cluster head that is currently involved in the traversal of another token. This condition is referred to as a *token collision*.

A cluster head is considered involved in the traversal of a token from the moment it created or received the token, until it receives the outcome of the balancing/synchronization process. The involvement of a cluster head in the traversal of a token consists of two phases: an *active phase* and a *passive phase*. As long as the token hasn't completed its traversal through the subtree headed by the cluster head (when taking the source of the token to be the root of the spanning tree), the cluster head is said to be in the active phase. During the active phase, the cluster head may be required, as part of the DFS traversal, to pass the token from one child to another. Furthermore, during the active stage, if the token has been balanced by a descendant of the cluster head, the traversal will result in a successful balancing process. If the token has traversed the subtree headed by the cluster head without being balanced, the cluster head has completed its active phase in the token traversal, and the traversal can result in either a balancing or a synchronization process. At this point the cluster head is said to be in the passive phase of its involvement in the token traversal.

Before describing how the multi-cluster algorithm is extended to handle token collisions, we would like to point out the following: a token collision is only possible when a token is passed to a cluster head that is in the active phase of its involvement in the traversal of another token. This observation can be easily explained by the following reasoning: assume that two tokens, denoted by Tok_1 and Tok_2, are currently traversing the network. Let the cluster head m_i be the first cluster head that received Tok_2 while being involved in the traversal of Tok_1. We denote by m_j the cluster head m_i has received Tok_2 from. Let us assume, by way of contradiction, that m_i is in the passive phase of its involvement in the traversal of Tok_1, therefore, Tok_1 has completed traversing the subtree headed by $s_{i,1}$, and specifically, Tok_1 has been passed to all of m_i neighbours, including m_j. In other words, m_j has been involved in the traversal of Tok_1 when it received Tok_2. This stands in contradiction to m_i being the first cluster head involved in the traversal of Tok_1 that has received Tok_2.

In order to handle token collisions, tokens hold two additional values: a unique identifier, identifying the source of the token, and the distance of the token from its source. Tokens will therefore be of the following form, $<\vec{a}, c, id, dist>$, where $\vec{a}$ and *c* denote the aggregation vector and token count, as in the original algorithm, and *id* and *dist* denote the identifier of the source of the token and the distance of the token from its source. Note that the distance of the token from its source can be easily maintained during the DFS traversal.

If a token collision has occurred, i.e., a cluster head, m_i, has received a token, Tok_2=$<\vec{a}_2, c_2, id_2, dist_2>$, while being involved in the traversal of another token, Tok_1=$<\vec{a_1}, c_1, id_1, dist_1>$, Tok_2 will be held by m_i until one of the following will happen: either m_i is notified that Tok_1 has been successfully balanced, or Tok_1 will be returned to m_i by one of its children. If Tok_1 has been successfully balanced, the traversal of Tok_1 has been terminated, thus resolving the token collision. If Tok_1 has been returned to m_i by one of its children, m_i will merge the two tokens.

Among the two tokens, the token that is closer to its source will be referred to as the *obsolete token*, and the other token will be referred to as the *dominating token* (token identifiers are used as tie breakers). The token collision will be resolved as follows: m_i merges the obsolete token into the dominating token. Assuming, with out loss of generality, that $dist_1 > dist_2$, the token resulting from the merge will be $<\frac{c_1 \cdot \vec{a}_1 + c_2 \cdot \vec{a}_2}{c_1 + c_2}, c_1 + c_2, id_1, dist_1>$. If the merged token is balanced, m_i will flood the merged token to all the cluster heads that were involved in the traversal of either Tok_1 or Tok_2, thus resolving the original constraint violations that lead to the creation of these tokens. If the merged token is not balanced, we would like it to continue traversing the spanning tree in a DFS search order rooted at the source of the dominating token. This can be achieved by sending the merged token to the source of the obsolete token, and continuing the traversal from there.

6 Relaxing the Precision Requirements

A desired trade-off when executing aggregate threshold queries is between accuracy and energy expenditure. Consider, for example, the temperature variance query given in Section 1. Say we are interested in receiving an alert when the variance in the temperature in a room exceeds 5 deg^2. In case the variance in the temperature is close to the threshold, drift vectors have very little leeway before the drift spheres they define are not monochromatic, leading to a high transmission rate. In the temperature variance query, it is sufficient to know that the variance in the temperature is *close* to the specified threshold. In other words, it is sufficient to require that the query returns a correct answer only when the variance in the temperature in the room is significantly far from the threshold value, say when it is smaller than 4.8 deg^2 or greater than 5.2 deg^2, but if the variance is very close to the threshold value (between 4.8 deg^2 and 5.2 deg^2), the query is not required to provide an accurate answer. This modified version of an aggregate threshold query enables efficiently detecting the desired alert condition, without expending expensive energy on keeping track of borderline values.

More formally, the modified aggregate threshold query is defined as follows: let $\vec{v}$ be the average measurement vector, as defined in Section 3, let *f* be an aggregation function, and let *r* be a predetermined threshold. Let ε be a predetermined error margin. We require that if $f(\vec{v}) > r + \varepsilon$, each mote will be able to determine that $f(\vec{v}) > r$, and that if $f(\vec{v}) \leq r - \varepsilon$, each mote will be able to determine that $f(\vec{v}) \leq r$. An aggregate threshold query conforming to these requirements is said to support an error margin of ε.

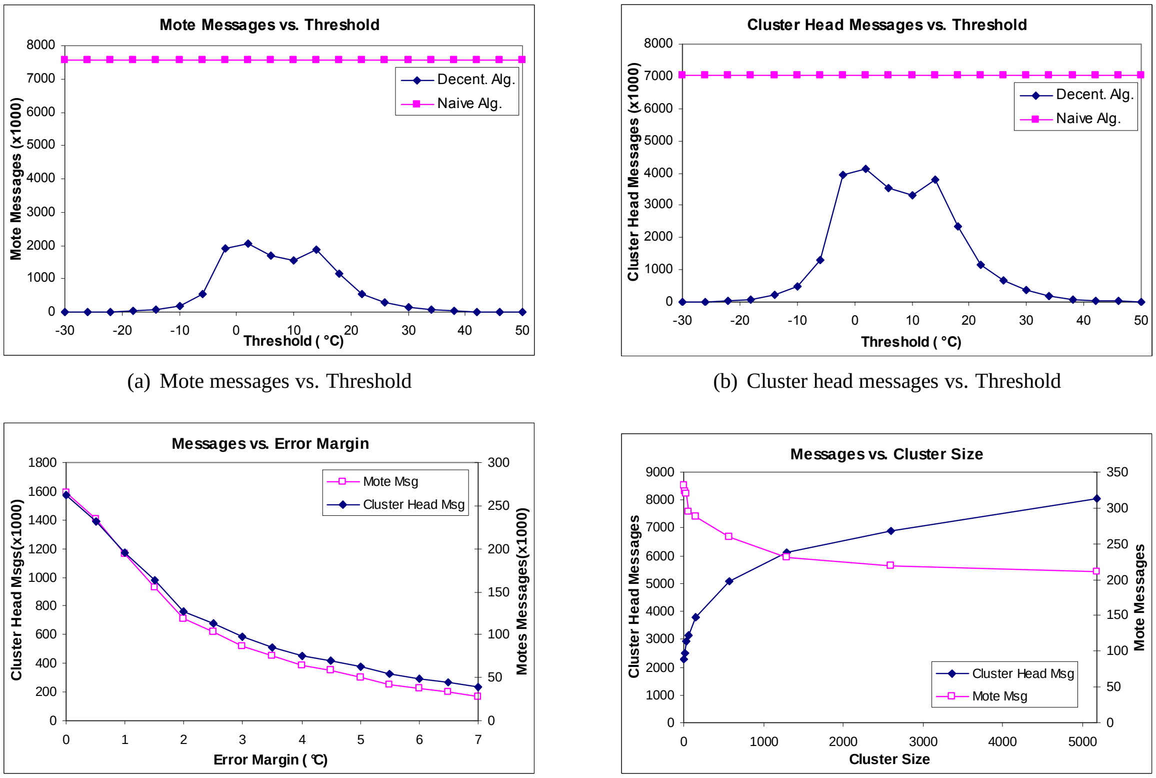
Our algorithm can be easily tuned to support an error margin of ε as follows: instead of working with a single coloring, induced by the aggregation function *f* and the threshold value *r*, two sets of coloring are defined, one induced by the aggregation function *f* and the threshold value $r + \varepsilon$, and a second induced by the aggregation function *f* and the threshold value $r - \varepsilon$. Whenever the original algorithm checks whether a ball is monochromatic, then, if $f(\vec{v}) \leq r$, the modified algorithm will check whether the ball is monochromatic according to the first coloring (the one induced by f and $r + \varepsilon$). If $f(\vec{v}) > r$, the modified algorithm will check whether the ball is monochromatic according to the second coloring (the one induced by f and $r - \varepsilon$). This ensures that if the value of the aggregation function on the estimate vector is below the threshold, and the value of the aggregation function on the average measurement vector is above $r + \varepsilon$ (or vice verse), the query will be reevaluated. At the same time, the modified algorithm ensures that regardless of the value of the estimate vector, the drift vectors have minimum leeway while maintaining monochromatic drift spheres, thus reducing the energy expenditure of the algorithm.

7 Experimental Results

We performed several experiments on real-world data in order to evaluate the performance of our algorithm. We simulated a network consisting of 5184 motes. The motes were positioned on a 144x36 grid. The motes were grouped into 81 clusters of 16x4. Each cluster was assigned a cluster head. In summary, the network consisted of a 9x9 grid of clusters, each cluster consisting of 64 motes.

We used climate data taken from [1] to simulate data measurements taken by the motes. The data set consists of temperature readings taken on a 144x72 grid that spans the entire globe. Temperature readings are taken at a resolution of 6 hours. We used data that corresponds to temperature reading covering the northern hemisphere over a period of a year, which yields a total of 144x36x1459 data measurements. According to the data, the average in the northern hemisphere ranges from -3.52 to 17 degrees Centigrade. The rational behind selecting this data set is that although the data set is collected on a global scale, it contains strong spatial and temporal correlations among data points, and is therefore similar in nature to what one would expect of a data set collected from a large scale deployment of a sensor network. Due to the lack of large scale real-world data sets taken from real deployments of sensor networks, we believe that this is the good alternative.

A spanning tree was constructed over the grid of clusters. We employed our algorithm in order to detect when the average temperature crosses a predetermined threshold. We compared the number of messages produced by our algorithm to the number of messages that would be generated by performing “in-network” aggregation, i.e. aggregating the drift vectors from the leaves upwards, and employing the aggregation function at the root. We refer to this algorithm as the *naive algorithm*. We make a distinction between mes-



(a) Mote messages vs. Threshold

(b) Cluster head messages vs. Threshold

(c) Messages vs. Error Margin

(d) Messages vs. Cluster Size

Figure 2. Experimental Results.

sages produced by motes, and messages produced by cluster heads. Clearly, since cluster heads are less resource constrained than motes, a message sent by a cluster head incurs a lower penalty than a message sent from a mote, but the exact difference between the cost of sending a message from a cluster head and the cost of sending a message from a mote strongly depends on the type of hardware used for motes and cluster heads. Therefore, for each experiment we present both the number of messages produced by motes and the number of messages produced by cluster heads.

In the first experiment, we ran our algorithm using various threshold values. No error margin was used in this experiment. Queries were run with threshold values ranging from -30 degrees Centigrade to 50 degrees Centigrade. Figure 2 shows the total number of mote messages and the total number of cluster head messages produced by our algorithm, as a function of the threshold value. In addition, we plotted the total number of mote messages and the total number of cluster head messages produced by the naive algorithm. Our algorithm significantly outperformed the naive algorithm for all threshold values, both in the number of messages produced by motes, and in the number of messages produced by cluster heads.

We expect that aggregate threshold queries will typically be used to detect anomalies, therefore, we were especially interested in the performance of the algorithm for threshold values that are close to the boundaries of the range of average temperature values. Since the data we used in our experiments is periodic in nature, the interesting threshold queries are the ones that detect when the average temperature diverges from its typical range of values, as opposed to queries that use a threshold that is within the typical range of average temperature values. When using threshold values that are close to the boundaries of the range of average temperature values, our algorithm outperforms the naive algorithm by orders of magnitude.

Next we checked the effect using an error margin has on the performance of our algorithm. We used a threshold value of -3 degrees Centigrade, and ran queries using error margins ranging from 0 to 7 degrees. Figure 2(c) shows the total number of mote and cluster head messages produced when using different error margins. As evident from the results, the error margin is very effective in reducing the number of messages produced both by motes and cluster heads. An error margin as small as 2 degrees Centigrade reduces the number of messages by more than half.

Finally, we conducted an experiment designed to examine the effect the choice of cluster size has on the properties of our algorithm. We ran a query with a threshold value of -3 and no error margin on the 144x36 grid of motes. We ran the query several times. Each run included all 5184 motes, but we used a different cluster size for each run. We used cluster sizes ranging from clusters of 4 motes, to a single cluster, containing all 5184 motes. In each run we recorded the average number of messages produced by motes, and the average number of messages produced by a cluster head.

Figure 2(d) shows the average number of mote messages and cluster head messages as a function of the cluster size. The results indicate that our algorithm performs better with larger cluster sizes. The number of mote messages decreases as the cluster size increases. We attribute the reduced number of mote messages in larger clusters to the fact that in larger clusters, intra cluster balancing is performed among motes that are more spatially dispersed, and therefore their measurements are more diverse, which enhances the efficiency of intra cluster balancing. Our results indicate that the number of cluster head messages increases as the cluster size increases, but one would expect the average number of cluster head messages to increase linearly in relation to the size of the cluster, since the number of constraint violations a cluster head needs to handle increases linearly in relation to the size of the cluster. Our results indicate that the number of cluster head messages increases sub-linearly in relation to the size of the cluster. We attribute the sub-linear increase in cluster head messages to the fact that as the the size of the cluster is increased, more constraints violations are resolved by the more efficient intra cluster balancing rather than by extra cluster balancing.

Note that while our results indicate that performance increases as the size of the cluster increases, the choice of cluster size is effected by additional considerations, such as the transmission range of cluster heads.

8 Conclusion

Continuous aggregate threshold queries are an important tool in a wireless sensor environment, since they are a natural candidates for expressing alert conditions. In many cases the exact value of the aggregate is not required in order to determine whether or not it has crossed the threshold, enabling the construction of algorithms that are far more efficient than algorithms that estimate these aggregates.

In this paper we presented an algorithm for performing continuous aggregate threshold queries over tiered sensor networks. The algorithm is based on a novel geometric approach, enabling the decomposition of the query into local constraints on the readings of the individual sensors. Experimental results on real-world data show that our algorithm reduces communications by orders of magnitude in comparison to a naive "in-network" aggregation approach.

References

[1] NOAA-CIRES Climate Diagnostics Center, Boulder, Colorado, USA http://www.cdc.noaa.gov/cdc/data.ncep.reanalysis.html.

[2] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE '04*, page 449. IEEE Computer Society.

[3] R. Govindan, E. Kohler, D. Estrin, F. Bian, K. Chintalapudi, O. Gnawali, S. Rangwala, R. Gummadi, and T. Stathopoulos. Tenet: An architecture for tiered embedded networks. *in CENS Technical Report 56*.

[4] M. B. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. In *PODS '04*, pages 275–285. ACM Press.

[5] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *DSN '01*, pages 433–442. IEEE Computer Society.

[6] W. Hu, N. Bulusu, C. T. Chou, S. Jha, A. Taylor, and V. N. Tran. A hybrid sensor network for cane-toad monitoring. In *SenSys '05*, pages 305–305. ACM Press.

[7] B. Krishnamachari, D. Estrin, and S. B. Wicker. The impact of data aggregation in wireless sensor networks. In *ICDCSW '02*, pages 575–578. IEEE Computer Society.

[8] R. Kumar, V. Tsiatsis, and M. B. Srivastava. Computation hierarchy for in-network processing. In *WSNA '03*, pages 68–77. ACM Press.

[9] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI '02*.

[10] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02*, pages 88–97. ACM Press.

[11] A. Manjhi, S. Nath, and P. B. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *SIGMOD '05*, pages 287–298. ACM Press.

[12] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04*, pages 250–262. ACM Press.

[13] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. In *SIGMOD '06*, pages 301–312. ACM Press.

[14] G. Sharma and R. Mazumdar. Hybrid sensor networks: a small world. In *MobiHoc '05*, pages 366–377. ACM Press.

[15] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04*, pages 239–249. ACM Press.

[16] T. Stathopoulos, L. Girod, J. Heidemann, and D. Estrin. Mote herding for tiered wireless sensor networks. *in CENS Technical Report 58*.

[17] H. Wang, D. Estrin, and L. Girod. Preprocessing in a tiered sensor network for habitat monitoring. *EURASIP JASP Special Issue on Sensor Networks*, pages 392–401.

[18] M. Yarvis, N. Kushalnagar, H. Singh, A. Rangarajan, Y. Liu, and S. Singh. Exploiting heterogeneity in sensor networks. In *INFOCOM '05*, pages 366–377.

[19] Y. J. Zhao, R. Govindan, and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *SNPA '03*.