# Location-integrated Indexing and Query Processing in Wireless Sensor Networks

BY

LIN XIAO
B.E., University of Science and Technology of China, 2002
J.D., University of Michigan, 2010

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2011

Chicago, Illinois

*To my family: Peng Liu, Lixia Jiang, Zhengming Xiao, Aijun Dang, Yongchuan Liu, Ping Liu, Yaohan Jiang*

*To my beloved grandparents: Kewen Zhu and Dengtang Xiao (1916-2008), without whom I wouldn't have been here at all*

## ACKNOWLEDGMENTS

Most important of all, I would like to thank my academic advisor Prof. Aris Ouksel for his wonderful guidance, consistent support and patience over the years. I would also like to thank my committee members Prof. Robert Sloan, Prof. Sol Shatz, Prof. Ajay Kshemkalyani, and Prof. Ashfaq Khokhar for their continuing support and advice. Additionally I would also like to thank Prof. Peter Nelson and Prof. Prasad Sistla, who were in my preliminary committee and oversaw the development of my thesis proposal. A number of faculty and staff at UIC were extremely helpful to me during my graduate studies, and I would like to thank them as well – particularly Prof. Ming Wang, Prof. Hui Lu, Professor Wenjing Rao, Howard Rockman and Santhi Nannapaneni. I would also like to thank the Office of Technology Management for their support throughout the last few months of my thesis work.

Thank you all for your wonderful guidance, support, and encouragement every step of the way.

LX

**TABLE OF CONTENTS**

**TABLE OF CONTENTS** (CONTINUED)

CHAPTER PAGE

**TABLE OF CONTENTS** (CONTINUED)

CHAPTER PAGE

## LIST OF FIGURES

CHAPTER PAGE

## LIST OF FIGURES (CONTINUED)

CHAPTER PAGE

## SUMMARY

Query processing on mobile sensor networks requires efficient indexing and partitioning of the data space to support efficient routing as the network scales up. Building an index structure in an ad hoc manner requires two operations: localization to discover the sensor's position; and assignment to each sensor of the appropriate data space partition.

We propose a fully distributed, cost-effective scheme, which integrates localization, indexing and data space management for sensors. The proposed scheme organizes the physical sensor network by localizing the sensors into physical zones, partitioning the data space into subspaces and assigning each subspace to a sensor for data management, and maintaining an overlay structure based on the space partitioning among sensor nodes and shared by queries.

# I. INTRODUCTION

Wireless ad-hoc sensor networks are being increasingly deployed in many applications, including structural health monitoring, habitat monitoring, traffic, and vehicle surveillance. For example, in a modern healthcare facility, doctors and nurses want to monitor the patients' daily activity and be informed whenever a patient needs emergency care. A sensor is attached to each patient so that the patient's pulse, blood pressure, movement and temperature can be sampled at a certain frequency. A sensor also checks the environmental conditions at the patient's location, like the humidity, noises, or air temperature. The sampled data is recorded by the sensor as events, which are stored and aggregated in the network. However, the patient's location may be unknown because the patient does not always stay in bed. The facility may offer outdoor places and reading rooms for the patient to relax. In addition, doctors may be interested in accessing abnormal fluctuations in a patient's temperature. A wireless mobile sensor network provides access to patient data regardless of the patient's location or availability. Such a network can significantly reduce the amount of time and cost to monitor patients, because regular manual checks can be replaced by sensor readings. With data gathered from sensors, regular reports may be sent to hospital staff, and the personnel in charge could be immediately informed of any emergency situations.

Data applications in wireless ad hoc sensor networks require scalable localization methods and dynamic data space partitioning among sensors to support efficient query processing. These challenges are exacerbated by sensors' inherent

constraints in storage capability, battery power, processing speed and communication bandwidth. To meet the imperative of energy preservation in data manipulation operations without performance degradation, we propose a scalable and self-configuring scheme, which synergistically integrates localization and data space partitioning, thus minimizing the otherwise prohibitive cost of separately performing localization and/or indexing.

Query processing in mobile sensor networks requires efficient indexing and partitioning of the data space to support efficient routing as the network scales up. Building an index structure in an ad hoc manner requires two operations: localization to discover the sensor's position; and assignment to each sensor of the appropriate data space partition.

In Chapter II, we propose a fully distributed, cost-effective scheme, which integrates localization, indexing and data space management for sensors. We show that without affecting the overall performance of routing and query processing, our localization scheme, which involves merely local interaction, is performed with significantly lower message and time cost than traditional approaches. The scheme, called "PRDS", enables sensors to self-configure into a consistent coordinate system and dynamically partitions the data space in a network with a small core of randomly scattered location-aware sensors within the geographical area spanning the network. We then propose, in Chapter III, an adaptive version of PRDS, which adds a refinement phase driven by a force based relaxation method called mass-spring optimization, to the basic scheme to handle cases where the fraction of location-aware

sensors is significantly reduced. Seamless transition is achieved between the basic scheme and refinement phase by assessing the current requirement of the network, while concomitantly, without user intervention, performing efficient local re-adjustment of the data space partitioning among neighborhood sensors. Our experiments, under a varying number of location-aware sensors, show that the "adaptive PRDS" achieves better results in many cases than the "simple PRDS" at the cost of a small increase in latency.

In Chapter IV, we propose an index-driven routing and query processing scheme and show its efficiency, robustness and scalability. We build an overlay structure, called the AggIndex tree, derived from indices computed by PRDS. With the index tree and underlying indexing structure, we also show how to efficiently perform query aggregation dynamically and efficiently along the tree paths. Updates to the AggIndex tree and to data space partitions only affect one to a few nodes. The AggIndex tree supports aggregation along tree paths while concentrating query searches to affected subareas in the network, thereby reducing energy consumption and latency. It also facilitates implementation of query aggregation optimization techniques such as query aggregation to further improve the performance of query processing. Analysis and experiments presented show that our approach significantly reduces query routing cost and latency compared to other approaches without incurring significant overhead for constructing and maintaining the overlay.

In addition, an integrated simulation platform (STARLET system) is designed and developed to study and compare algorithm performances in the areas of

localization, event handling, network topology control, routing, and query processing in various types of sensor networks. The design of the simulator is presented in Chapter V.

When there is no ambiguity, we sometimes refer to a sensor as a node.

## II. TOLERANCE OF LOCALIZATION IMPRECISION IN EFFICIENTLY MANAGING MOBILE SENSOR DATABASES

Query processing on mobile sensor networks requires efficient indexing and partitioning of the data space to support efficient routing as the network scales up. Building an index structure in an ad hoc manner requires two operations: localization to discover the sensor's position; and assignment to each sensor of the appropriate data space partition. We propose a fully distributed, cost-effective scheme, which integrates localization, indexing and data space management for sensors. We show that without affecting the overall performance of routing and query processing, our localization scheme, which involves merely local interaction, is performed with significantly less message and time cost than traditional approaches.

### A. Introduction

In wireless ad hoc sensor networks, it is established that data indexing structure can greatly improve query processing efficiency [1]. Two basic operations are required for this purpose: *localization* and *routing*. Localization makes a sensor node aware of its own position and neighborhood, and in turn, enables systematic mapping of dynamically obtained position information onto the data space partition managed by the node. Routing determines at each step the optimal next hop to transmit messages from a source to a destination. Its goal is to minimize latency.

Each sensor must maintain its location information to enable the delineation of its zone boundaries in an environment of frequent topology changes. Sensors can be equipped either with high-precision GPS **(**Global Positioning System**)** receivers, or with components capable of performing distributed localization. Localization imprecision may vary from centimeters to tens of meters depending on the specific localization algorithm utilized. Highly precise algorithms have, in many cases, been proven to be either computationally impractical, or prohibitively expensive due to high message or computation costs, and expensive devices.

GPS is regarded as the most popular satellite navigation system for localization and navigation. However, its expense, huge energy consumption, and bulkiness often preclude its deployment at every node of a large wireless sensor network. Further, in many outdoor urban areas and most indoor environments, GPS is not available and/or its capabilities are severely curtailed. For example, in metropolitan area where the satellite signal is blocked by high-rise buildings, or in indoor environments such as basements, the GPS device is often unable to provide localization [2].

Localization in sensor networks and wireless ad hoc networks has been explored extensively over the past decade. Many of the proposed techniques achieve highly precise localization result, but at the cost of substantial number of messages. The minimum average cost is achieved by the beacon-based approaches, which is at least O(*Bnd*), where *B* is the number of beacons, *n* the network size and *d* the network

diameter [3]. The other approaches are generally more costly. A summary can be found in the "related work" section of this Chapter. The cost is particularly significant in mobile environments where it is necessary to frequently re-localize the nodes [4].

This raises the question as to whether fine granularity precision is necessary, and if not, what is the minimum precision level required to support indexing, and thereby, efficient routing and querying?

Assuming a reasonably small core of location-aware sensors in a WSN, whose positions are obtained either from GPS receivers or fixed point pre-measurements, our contention is that highly precise sensor localization is not always necessary in database applications. Using our indexing mechanism, it suffices to efficiently localize a sensor within a clearly delineated zone in a large network. This presupposes the availability of a scalable distributed indexing method, which supports locally self-configuring zone boundary delineation. Since highly precise localization is costly, those schemes that do not require an excessive level of precision, yet providing efficient data manipulation operations will obviously improve the overall performance of the network. Our localization-integrated indexing algorithms are designed with this goal. It requires only very limited initial location information in the whole network.

We argue that those stand-alone approaches are not necessary to build indexing structures for efficient query processing in wireless ad hoc sensor networks. In fact, these approaches may have a significant adverse effect on performance. Their

excessive messaging will exacerbate congestion. In addition to that, their computation requirements for set-up, stabilization and adjustment expand greatly the demand on power and bandwidth consumption. This in turn imposes negative compounding effects on query processing performance, particularly in situations where the indexing structure must be frequently adjusted due to node movements, and thereby causing the amortized query processing cost to increase substantially.

We propose a scheme, which integrates localization, indexing and data space partitioning. This is done by first determining through local interactions the *position region*, which is the tightest computed region enclosing a node; second, by mapping this position region to a multidimensional data space partition based on our indexing policy. To the best of our knowledge, this work is the first to propose such an approach which achieves this objective and does so efficiently. Analysis and experiment show that our scheme uses fewer messages, tolerates more localization imprecision, and yet requires fewer updates in the indexing structure than traditional approaches and enables efficient routing and query processing.

## B. Related Works

Localization approaches in sensor networks have been extensively explored over the past few years, such as [5] [6] [7] [4] [8] [9] [10]. Summaries and evaluations can be found in [11] [12]. Techniques such as DV-hop, DV-distance, or Euclidean scheme are used to get a first approximation of the position relative to a beacon or anchor node with known position, then multi-lateration [4] or Kalman filter [6] or Bayesian filter [9] is applied to improve the estimation. Many of these approaches achieve very precise localization result, but incur an excessive number of messages. In the case of mobile nodes, the cost of re-localization is very significant [4].

An analysis of beaconing-based approaches, such as that in Calamari [13], indicates a message complexity of at least $O(BNd)$, or for more accurate measurement, $O(N^2)$, where $B$ is the number of beacons, $N$ is the network size and $d$ is the network diameter. For each node, at least O($Bd$) or O($N$) messages are required on average.

The only literature we found bearing some similarity to ours is in [8], which also uses bounding rectangle for location calculation. Our approach differs from [8] in the following ways: (1) we use ranging distance in the calculation while [8] is based on pre-defined fixed range when two sensors are within each other's range. As a result, we obtain more precise measurements, as our experiments show; (2) our position region and data space partitioning algorithm apply to moving nodes while [8] does not offer any clear solution; and (3) our approach integrates both localization and data partitioning, whereas [8] is restricted to localization only. We will also show that in

our structure, one node only needs to send O(*deg*c*) messages to calculate its position region, where *deg* is the out-degree of the node. This is significantly lower than any of the previous approaches.

Distributed multidimensional indexing structures make it possible to efficiently answer queries including exact-match query, multidimensional range query, and aggregations. Multidimensional indexing structures have been extensively studied in the traditional database society in past twenty years. A literature survey can be found in [5]. Later, researchers have revisited the indexing problem in a distributed environment first in P2P networks with examples including CAN [14], CHORD [15] and G-Grid [16]. P2P networks are different from sensor networks in that P2P networks are usually computers connected via the Internet and therefore point-to-point connection can be established directly between any two IP addresses, while on the other hand sensor networks have more physical constraints, such as node connectivity, limited communication range, limited bandwidth, physical locations, and energy preservation etc.

When sensor networks became a major research area at the beginning of the 21st century, ad-hoc sensor networks with indexing structures such as GHT (Geographic Hash Table) [17] and DIMS [1] were proposed. DIMS [1] used a locality-preserving hash function to store data, thus enabling multidimensional range queries. GHT [17], on the other hand, used consistent hashing, where data are hashed to a specific point and managed by the closest sensor to that point. Our proposed approach bears more similarity in data management to DIMS in that each node maintains a zone, and data regions are mapped to corresponding zones with locality

preserving hash functions. Another distinction is that both GHT and DIMS require exact position information known to every node in the network. Therefore, our approach differs from DIMS in that exact position information is not needed: localization is performed online and at the same time a more flexible indexing structure is dynamically constructed across the network in a manner that allows fewer updates to the indexing structure as sensors move around.

Our work in this chapter has been published in [18] and also covered under US Patent No. 7,840,353 [19]. Later, [20] proposed a scalable localization solution for underwater sensors with mobility predicting functions, where mobility was calculated from past moving patterns of mobile sensors and more precise location information gathered from a set of anchor/reference nodes.

### C. Definitions and Data Models

Our approach maps the multidimensional data space onto the two-dimensional space, and then partitions the latter space into rectangular zones. For simplicity, we shall assume that the sensor network environment covers a bounded two-dimensional rectangular area. The extension to a three-dimensional area is straightforward.

Formally, consider a relation *R* with attributes $A_1, A_2, \cdots, A_d$, taking their values from bounded but not necessarily finite domains $D_1, D_2, \cdots, D_d$ respectively. A normalization of the multidimensional data space into $[0,1)^d$ maps each data point *R* to a *d*-dimensional vector $b = (b_1, b_2, ..., b_d)$ where each value $b_i$ represents the mantissa of

the corresponding mapped attribute value. Let sequence $b_{i,1}b_{i,2}...b_{i,\max}$ be the binary representation of $b_i$, where *max* is the maximum possible length for all $b_i$'s. Utilizing the order-preserving hashing function previously defined for multidimensional structures IBGF [21], NIBGF [22] [23], and G-Grid [16], a unique key is obtained for each data point $R$ by interleaving the bits of the binary sequences cyclically as follows:

$$K = K_1 \cdots K_{d*\max} = b_{1,1}b_{2,1}\cdots b_{d,1}\cdots b_{1,\max}\cdots b_{d,\max} \qquad (1)$$

This key *K* is mapped into two-dimensional space as follows:

$$\begin{aligned} K_x &= k_1 k_3 \cdots k_{2*\left\lceil \frac{d*\max}{2} \right\rceil - 1} \\ K_y &= k_2 k_4 \cdots k_{2*\left\lceil \frac{d*\max}{2} \right\rceil - 2} \end{aligned} \qquad (2)$$

where $K_x$ and $K_y$ are the *x* and the *y* components of key *K* respectively.

This mapping is *locality preserving* in that the relative positions of the points in the multidimensional data space are maintained in the two-dimensional space. This property is further stated as a theorem in Chapter IV.

A ***data region*** is the *d*-dimensional hypercube defined by a unique *identifier I*. Let the binary sequence representing *I* be $i_1 i_2 \cdots i_l$ . Data whose keys Ks are prefix of *I* are mapped into the data region identified by *I*. The length *l* of the identifier, also called the *level* of the data region, is the number of times the initial data space $[0,1)^d$ has been split to obtain the data region.

A data region is split in half cyclically along all its dimensions. Each time a split occurs, two new data regions are created. Their identifiers are obtained by appending 0 or 1 to the identifier of the split data region. The original space is assigned identifier "0". A multidimensional data region *D(I)* with identifier *I* is bounded by

$$[L_1,U_1)\times[L_2,U_2)\times\cdots\times[L_d,U_d) \qquad (3)$$

where $L_i=\frac{x_i}{2^{l_i}},U_i=\frac{x_i+1}{2^{l_i}}$, and $l_i,1\le i\le d$ is the number of times this data region has been split along the *i*-th axis, denoted by $l_i=\lceil (l-i+1)/d \rceil$. The *d*-tuple $x=(x_1,x_2,\cdots,x_d)$, $0\le x_i<2^{l_i}$ gives the integer coordinates of the data region calculated as following:

$$x_j=i_j*2^{l_j}+i_{j+d}*2^{l_j-1}+\cdots+i_{j+(l_j-1)*d}*2^0 \qquad (4)$$

A data region *D*(*I*) is mapped to a region in the two-dimensional space $[0,1)^2$, referred to as a ***zone*** with the same identifier *I* and denoted *Z(I)*. Similarly, the level of a zone is also defined as the length of the binary representation of the zone's identifier. A zone *Z*(*I*) is given by:

$$[L_x,U_x)\times[L_y,U_y) \qquad (5)$$

where $L_x=I_x/2^{l'},U_x=(I_x+1)/2^{l'}$, $L_y=I_y/2^{l-l'},U_y=(I_y+1)/2^{l-l'}$, $l'=\left\lfloor\frac{l}{2}\right\rfloor$. $I_x(n)$ and $I_y(n)$, calculated from (2), are the *x* and *y* components of *I*. $l'$ and $l-l'$ are their lengths respectively.

The mapping between data region *D*(*I*) and *Z*(*I*) is one-to-one. Figure 1

illustrates an example of zone partitioning in the two-dimensional space. In the three-dimensional data space, data region 110 represents data in the range of $[\frac{1}{2},1)\times[\frac{1}{2},1)\times[0,\frac{1}{2})$ , according to (3). Correspondingly, the two-dimensional space covered by zone 110 is delineated by: $[\frac{1}{2},\frac{3}{4})\times[\frac{1}{2},1)$. The level of both the data region and the zone 110 is 3.

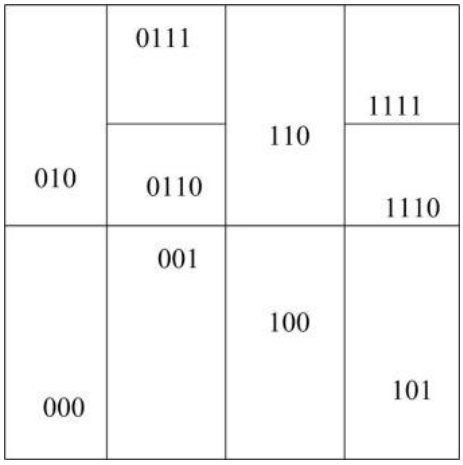



**Figure 1 Multidimensional data space mapped to two-dimensional space.**

Every data region and its corresponding zone are assigned to a single sensor node, which is then responsible for its management. The creation of a new zone, and its mapping to a new sensor node *q* are triggered by node *q* arriving in the neighborhood of a node *p* already existing in the system which manages zone *Z*(*I*). A node *p* is node *q*'s ***neighbor*** if *q* is within *p*'s direct communication range. Zone *Z*(*I*) is split into two zones *Z(I')* and *Z(I'')*, where *I*' and *I*'' are obtained by appending 0 or 1 to the binary sequence of identifier *I* respectively. The relative position of the new

node *q* with respect to *p* determines how the assignment is performed. We show below that the position information of the new node *q* is calculated based on local interaction with its neighbors including *p*.

***Position region*** of a node *n*, denoted *PR*(*n*), is defined as the area spanning the likely positions of node *n*. It is calculated based on the distances of node *n* to its neighboring nodes. Since the exact positions of the neighbors may not be known, their positions are approximated by their current position regions. The distance is obtained by ranging, which measures the distance between two nodes within transmission range. RF-based and acoustic ranging are the two most popular classes of ranging techniques [5].

Let $n_i$ be the *i*-th neighbor of a new node *n* and $d(n,n_i)$ be the measured distance between *n* and $n_i$. We define $PR(n, n_i)$ as the position region of *n* relative to $n_i$, given $d(n,n_i)$. For simplicity, we assume that the sensor network space is represented by $[0,m)\times[0,n)$. A zone is mapped into a rectangular area of the sensor network space, called the ***physical-zone***. For example, in Figure 1, zone 110 covers the two-dimensional space $[\frac{1}{2},\frac{3}{4})\times[\frac{1}{2},1)$ in the zone space. Its physical-zone will be $[50,75)\times[50,100)$ in a sensor network's physical space $[0,100)\times[0,100)$.

The ***position region coverage*** of neighbor $n_i$ of node *n*, denoted $PRC(n,n_i)$, is defined as the intersection of *n*'s position region $PR(n)$ and $n_i$'s physical-zone $PZ(n_i)$ in the two-dimensional space. Figure 2 illustrates an example: $PRC(n,0111)$ is shown

in the as the solid line rectangle. Obviously, it is equal to $PR(n)$ because $PR(n)$ has no intersection with other physical-zones.

### D. Localization

The exact geographic information of a node may not be available. Therefore, its current position can only be estimated. Our approach uses all available position information in the network for this estimation. The sources could be directly from GPS or Assisted-GPS, or indirectly from its interaction with other devices whose location is known. A sensor may also be static, in which case its position can be pre-measured.

Traditional beacon/anchor methods [24] [25] inform the whole network of the beacon/anchor positions by flooding. In this way, every node obtains an initial estimate of its distance to all the beacon/anchor nodes. A summary and performance comparison of beacon/anchor approaches can be found in [26]. Our localization method avoids flooding, as the nodes are not directly informed of the beacon/anchor nodes' locations. Nodes depend solely on the position information provided by their direct neighbors, as explained in the definition of the position region. As a result, our method is scalable as the number of messages is significantly reduced. This is particularly important in mobile environments where position readjustment is frequent. New position information may also be exploited at any time as it becomes available, allowing progressive insertion of new position information into the system. For example, when new sensors are added or are reconnected to the system.

Our algorithm proceeds as follows: a new node without position information first initializes its position region to be the whole space. Then, after obtaining ranging distance with each neighbor, it determines the rectangle bounding both the node and the position region of this neighbor. The new position region is then the intersection of all the bounding rectangles formed with its neighbors. If all the neighbors of a node have stable position regions, then this newly computed position region will also be stable. If, on the other hand, the neighbors' position regions are instead transient, then the computation of the new node's position region will itself continuously require refinement. A node's convergence to a stable position region depends on the dynamics of all the neighbors. For every type of network, there is an acceptable threshold in the number of iterations beyond which the position region is considered "reasonably" stable, i.e., the difference between the current position region and the position region at the limit is bounded by the threshold. The data space partitioning phase proceeds after stability is decided.

Let $PR(n)$ be represented by $[PL_x(n), PU_x(n)) \times [PL_y(n), PU_y(n))$. The following algorithm calculates position region $PR(n)$:

**Algorithm II.D.1 Position_Region_Discovery ($n$)**

- Step 1: If node $n$ is aware of its own position $(p_x, p_y)$, its position region is trivially $[p_x, p_x) \times [p_y, p_y)$ then EXIT else proceed to Step 2.
- Step 2: Initialize $n$'s position region to the whole physical space $[0, m) \times [0, n)$.

- Step 3: For each neighbor $n_i$ of $n$, find $PR(n,n_i)$ and intersect it with the whole physical space:

$$PR(n,n_i) = [PL_x(n_i) - d(n,n_i), PU_x(n_i) + d(n,n_i)) \times [PL_y(n_i) - d(n,n_i), PU_y(n_i) + d(n,n_i)) \bigcap [0,m) \times [0,n)$$

- Step 4: $PR(n) = \bigcap_{i=1}^{\deg(n)} PR(n,n_i)$

**End PRD**.

Figure 2 illustrates the computation of *PR*(*n*). The blackened node in this figure represents node *n*. It has three neighbors identified 010, 0111 and 0110 respectively, represented by hollow nodes. The shadowed rectangle around each neighbor shows its *PR*. The solid line rectangle around *n* represents *PR(n*).

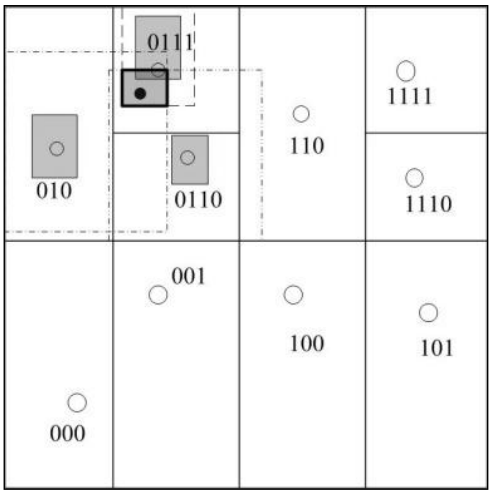



**Figure 2 The position region for a newly joined node *n***

### E. Data Space Partitioning

After calculating $PR(n)$, one of $n$'s neighbor is selected to have its zone $Z$ split and one of the partitions assigned to $n$. The following algorithm determines this neighbor and performs the appropriate data space partitioning.

**Algorithm II.D.2 Data_Space_Splitting ($n$)**

---

Step 1: For each neighbor $n_i$ of $n$, $PRC(n,n_i) = PR(n) \wedge PZ(n_i)$, select the $n_i$ with largest $PRC(n,n_i)$.

Step 2: Split $n_i$'s zone $Z(I)$ until two zones $Z(I_1)$ and $Z(I_2)$ intersecting $PR(n)$ and $PR(n_i)$ are obtained.[1]

Step 3: If each of $PR(n)$ and $PR(n_i)$ are fully enclosed in $Z(I_1)$ and $Z(I_2)$,

then assign $n$ and $n_i$ the enclosing zones $Z(I_1)$ and $Z(I_2)$, respectively. Exit.

else find a reference node $r$ with respect to $n_i$ (see below for the definition of "reference node"). [2]

Step 4: Find the relative position of $n$ with respect to $n_i$ along the splitting axis.[3]

---

[1] Recall in step 1 $n_i$ has been chosen as the splitting node. Since a zone is split in half cyclically along the $x$-axis and $y$-axis, the direction of split is determined by $Z$'s current level $l$. If current $l$ is even, the next split is done along the $x$-axis; otherwise, it is along the $y$-axis.

[2] This step is necessary when zone assignment to the nodes is ambiguous. This occurs when the two nodes cannot distinguish their relative positions with respect to the splitting axis. Specifically, there is ambiguity when the projections of $n$'s and $n_i$'s positions regions onto the splitting axis intersect.

[3] Nodes $n$, $n_i$ and $r$ form a triangle. Let $\alpha = \angle nn_ir$. The edges $\overline{rn_i}$ and $\overline{nn_i}$ are measured by ranging as $d(r,n_i)$ and $d(n,n_i)$. If $\overline{nr}$ can be measured directly, then $\alpha$ can be calculated easily, otherwise $\overline{nr}$ is approximated by the distance between the centers of $PR(n)$ and $PR(r)$.

Step 5: Assign $Z(I_1)$ and $Z(I_2)$ accordingly.[4] **End DSS.**

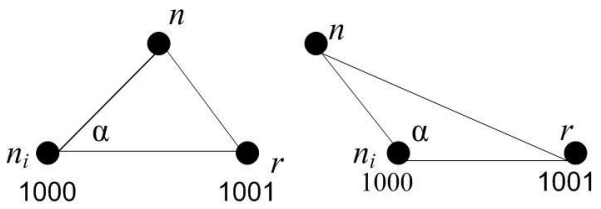



**Figure 3 Examples of references nodes**

Observe that in Figure 2, after applying step 1 of this algorithm, the new node *n* chooses to split with node 0111. But *n*'s *PR*(*n*) intersects with *PR*(0111), therefore their projections onto the splitting axis also intersect. Since their relative positions

---

Because $\cos\alpha = \frac{d(r,n_i)^2 + d(n,n_i)^2 - d(n,r)^2}{2d(r,n_i) * d(n,n_i)}$,

$d(r,n_i)|^2 + |d(n,n_i)|^2 - |d(n,r)|^2 \le 0$ shows $\alpha \le 90^\circ$, otherwise $\alpha > 90^\circ$.

If $\alpha \le 90^\circ$, *n* and *r* are on the same side of $n_i$ when projected onto the splitting axis. Otherwise, they are on opposite sides. Figure 3 shows two examples of ambiguity resolution, where *x* is the splitting axis, 1000 is the node $n_i$ whose zone will be split and 1001 is its right reference node *r*. In Figure 3.a, $\alpha \le 90^\circ$, node *n's* projection on *x*-axis is thus to the right of $n_i$. The opposite case is shown in figure 3b where $\alpha > 90^\circ$, *n*'s projection on x-axis is to the left of $n_i$.

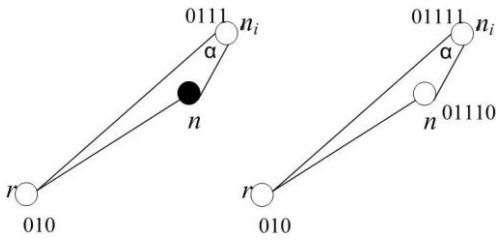
[4] The node whose projection on the splitting axis is to the left of the other is assigned zone $Z(I_1)$ and the other is assigned $Z(I_2)$. For example, in Figure 3a, $n_i$ manages zone 1000, which will be split into 10000 and 10001 at the arrival of new node *n*. Node *n* is determined to be to the right of node 1000 when projected on splitting axis x. It will therefore be assigned zone 10001 whereas node $n_i$ will get zone 10000.

Figure 4 is based on Figure 2. It shows new node *n*'s splitting with node 0111. Node 110 is a left reference. The split is along the *y*-axis. According to our algorithm, *n* discovers that the angle $\alpha \le 90^\circ$, meaning that *n*'s projection on *y*-axis is the same side as reference node 110 with respect to node 0111. As a result, node 0111 is assigned zone 01111 whereas node is assigned 01110.

cannot be decided merely from their *PR*s, there is ambiguity. A reference is then needed to resolve this situation. Node 010 is the one selected as reference in this case. The ambiguity is resolved by finding $n_i$'s reference node *r.* Reference node *r* is chosen from $n_i$'s neighbors $n_{i_j}$ by the following criteria: $I_{axis1}(r) \neq I_{axis1}(n_i)$ and $\left|I_{axis2}(r) - I_{axis2}(n_i)\right| = \min\left|I_{axis2}(n_{i_j}) - I_{axis2}(n_i)\right|$ among all $n_{i_j}$ where *axis1* and *axis2* refers to the splitting axis and the non-splitting axis respectively.

Intuitively, *r* is chosen among those $n_i$'s neighbors whose projection along the splitting axis is distinct from $n_i$'s and distance along the non-splitting axis is minimum.

The message complexity of the DSS algorithm is O(*deg*), where *deg* is the maximum out-degree for a node, since each neighbor only needs to send one message to the new node informing it of its current position region and its identifier. The time complexity is O(1), because the algorithm requires only five steps at most.




**(a) Before splitting (b) After splitting**

**Figure 4 Before and after splitting**

In the worst case, The PRD algorithm has message complexity of O(*deg***c*) and time complexity of O(*c*). Here *c* is the threshold set on the maximum number of iterations to stabilize the position region. Our experiments show this number to be very low. Thus both algorithms save greatly on the energy and power consumption.

**Discussion 1** Different policies may be deployed in selecting the node whose zone will be split. The policy adopted in the PRD algorithm is as follows:

If *n*'s position region is fully contained in neighbor $PZ(n_i)$, meaning $PRC(n,n_i) = PR(n)$ and $PRC(n,n_k) = \varnothing$ for all $k \neq i$. Then node $n_i$ is definitely selected for splitting with node *n*.

If *n*'s position region has intersection with multiple physical-zones, for all *k* such that $PRC(n,n_k) \neq \varnothing$, choose the node $n_i$ with the maximum position region coverage $PRC(n,n_i)$ as the node to split zone with node *n*. □

The policy described above selects the neighbor node $n_i$ whose physical-zone has the highest probability to include the new node *n*. An alternative is to select the neighbor with the largest zone area still intersecting *n*'s position region. In the latter case, the zone may be larger than node $n_i$'s communication coverage. It is then judicious to split this area with other nodes, and thereby enhances the probability of balancing the space of communication coverage among the nodes. Yet another policy is to choose the zone with the largest data load satisfying the splitting requirements. This will help balance the load and relieve hotspot situations. Finally, a hybrid approach is one that combines all the three approaches into a single fine-tuning

algorithm that will enable balancing the trade-offs between load, space coverage, and/or geographic information preservation, according to the need of the application and the distribution of data.

In the case of a mobile node, readjustment will be invoked when its current position region becomes disjoint from the zone it manages. This will result in less frequent updates to the indexing structure. We refer to this approach as “lazy updating”. For example some nodes tend to move frequently within a restricted area, “lazy updating” will avoid a lot of unnecessary back-and-forth updates while at the same time preserving efficiency.

## F. QUERY PROCESSING

We will only briefly discuss query routing and processing in this section. The main part of the work on query processing is to be presented in Chapter IV.

Each node maintains a list of its neighbors' identifiers. If a query is a point query, the node will calculate the key for that data item. If a query is a range query, the common prefix of keys in this range is calculated. The query then is directed to all zones with the common prefix.

Each node, on receiving a single data query, will check if its current zone identifier is a proper prefix of the data's key. If so, it is the destination and will return the query result to the source node. If it is a range query, the node will check if its current identifier is a proper prefix of the common prefix of keys, or the common prefix is a proper prefix of its current identifier. If this is the case, the destination of the query has been reached and the result is returned to the source node. If the node is not a destination, it will forward the query to the neighbor node whose identifier is closest by Euclidean distance to the destination. If the node itself is closer than any of its neighbors, it will use iterative BFS to find the closer node in its neighborhood.

## G. ANALYSIS

We classify localization error in PRD into two types:

a. First-class error (outbound error). This error is the distance of the actual position of a node from its position region (*PR*). If the actual position is properly contained in its *PR*, the first class error is zero. The first-class error represents the ***accuracy*** of the localization approach.

b. Second-class error (inbound error). This error indicates how far the node's actual position is from its *PR* borders. It can be represented by the area of *PR*. The second-class error represents the ***granularity*** of the localization approach.

As in most literature [27], we assume the measurement noise is white Gaussian (Gaussian with zero mean). The standard deviation of measurement noise is also assumed to be proportional to the distance $d(i, j)$ between two sensors $i$ and $j$. Let $e_0$ be the variance of the measurement error over a unit distance. Then the measurement noise $e_{i,j}$ between $i$ and $j$ is given by:

$$p(e_{i,j}) = N(e_{i,j}; \mu, \sigma^2), \quad \mu = d(i, j), \sigma^2 = d(i, j)^2 e_0^2 \tag{6}$$

Figure 5 illustrates the accumulated measurement errors of position information propagation from node *c* to *a* through *b*. Node *c* is aware of its own position. The left border of *PR*(*a*) is computed from the left border of *PR*(*b*), which in turn is computed from *c*'s position. The distances between a and b, and between *b*

and $c$ are measured as $d(a,b)$ and $d(b,c)$ respectively. Path $c \to b \to a$ is shown in bold in figure 5. Measurement noise in each hop is independently white Gaussian, according to (6).

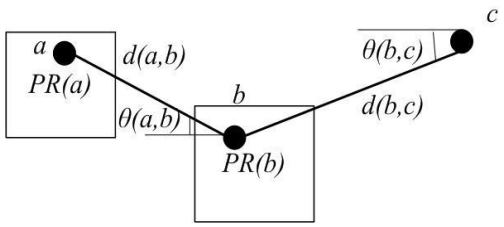



**Figure 5 An example of position information propagation.**

Generally, assume position information propagation from node $n_1$ reaches node $n_k$ in $k$ hops. Let $d(n_i,n_{i+1})$ be the measured distance between node $n_i$ and $n_{i+1}$ for $1 \le i < k$. The accumulated measurement noise is also white Gaussian. Its probability density function (pdf) is given by:

$$
\begin{aligned}
p(x) &= N(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \\
\sigma^2 &= \sum_{i=1}^{k} \sigma_i^2 = \sum_{i=1}^{k} d(n_i,n_{i+1})^2 e_0^2, \\
\mu &= \sum_{i=1}^{k} d(n_i,n_{i+1})
\end{aligned}
\qquad (7)
$$

Let's consider now the second-class error in Figure 5. Let $\theta(a,b)$ be the angle between vector $\overrightarrow{d(a,b)}$ and the $x$-axis, and $\theta(b,c)$ the angle between $\overrightarrow{d(b,c)}$ and $x$-axis. The

calculation of $PR(a)$ from $d(a,b)$ will put the left boundary as far as $d(a,b)$ from $PR(b)$'s left border. However, the actual position of $a$ is only $d(a,b) * \cos\theta(a,b)$ apart from b along $x$-axis. Thus, the second-class error associated with this step is then their difference $d(a,b) *(1- \cos\theta(a,b))$.

Generally, let the angle between $d(n_i,n_{i+1})$ and $x$-axis be $\theta(n_i,n_{i+1})$ for $1 \le i < k$ . Then the accumulated second-class error at $n_k$ is given by:

$$\sum_{i=1}^{k} d(n_i,n_{i+1}) * (1-\cos\theta(n_i,n_{i+1})) \qquad (8)$$

One observation is that in most cases, the measurement error (1% − 5%) is only a small portion of the second-class error. Therefore its effect is negligible and can be omitted from the second-class error analysis. However, in the rare cases where the second-class error is lower than the accumulated measurement error, a first-error may occur. Intuitively, this happens when the nodes on the propagation path are almost aligned on a horizontal line or vertical line. In this situation, the calculation of $PR(n_k)$ could possibly fail to include node $n_k$. We will see in our experiments that first-class errors happen very rarely in randomly generated graphs.

## H. EXPERIMENT EVALUATION

### 1. Simulation Setting

We set up an extensive simulation model to measure the performance of our **PRD**/**DSS** algorithm. The network is an arbitrary $l*w$ rectangle area. For simplicity and without loss of generality, we use a square area where $l = w$. Let $N$ be the

network size, i.e., the number of sensors in the network, and *com* be the communication radius for each sensor. The network of sensors is generated as a random graph with uniform node distribution.

We tested on *N* varying from 50 to 2,000. The average number of neighbors per node, i.e., the average out-degree per node, represents the density of the network. We also vary the density to see its effect on the performance.

In our simulation, the noise of ranging measurement is characterized as white noise and is assumed to be a Gaussian distribution with mean 0. The standard deviation of error is a linear function of distance and unit noise bound $e_0$. Depending on the particular ranging technology adopted in sensing devices, in our simulation we set the noise bound $e_0$ to be 1%, 5%, 10%.

Sensors may have geographic information. In our experiments, we define a variable parameter, denoted f, to represent the ratio of nodes with geographic information over the total number of nodes.

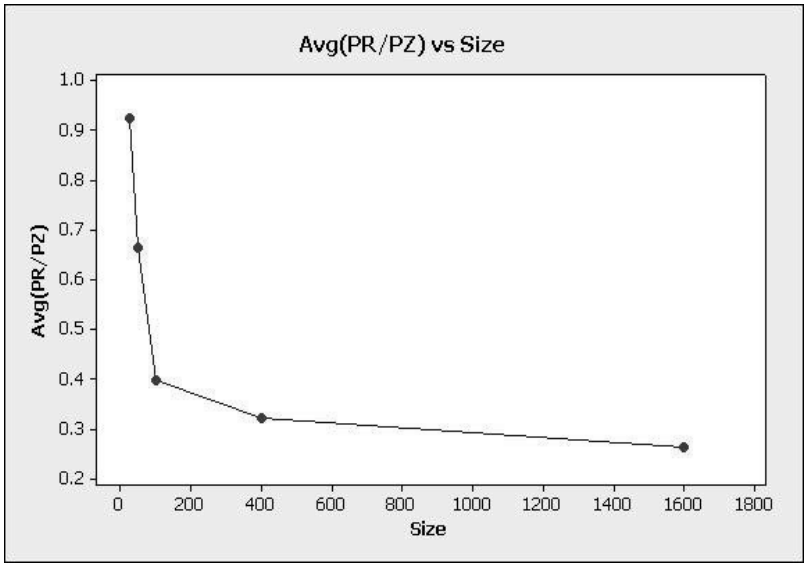
Sensors may also be static or mobile. For a mobile sensor, a random number is generated in each iteration to determine the motion direction. A sensor has equal probability to move in any of the four directions: left, right, up and down. Each step is of the same length, say 2 meters. This simulates a slow random walk in 2-D space.

We have conducted analysis of different combinations of levels of available geographic information and sensor movement patterns. In each, both first-class errors

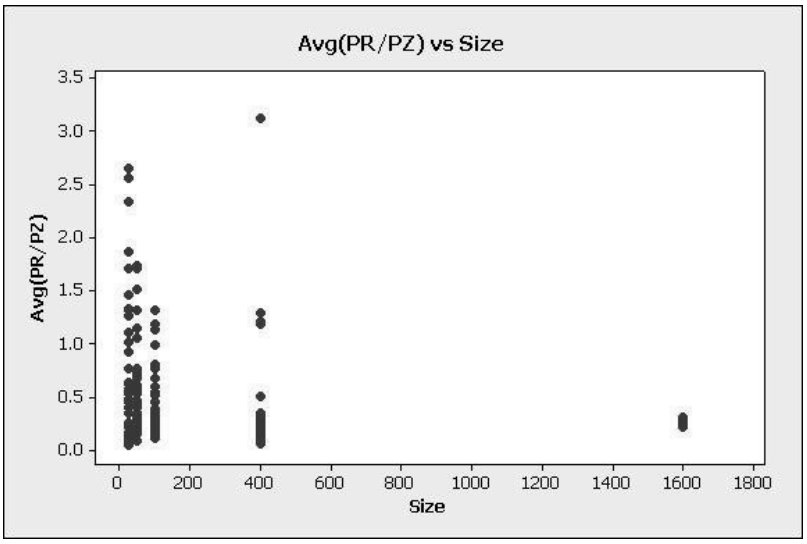
and second-class errors were investigated. First-class error is determined by evaluating the probability that a node's actual position is out of its *PR*. The second-class errors were analyzed by evaluating the average ratio of a position region divided by physical zone size Avg(PR/PZ).

### 2. Effects of Network Size

Fixing the density, noise bound $e_0$ and ratio f, we increase the network size by doubling the number of nodes N from 25 to 1,600, and for each N, we generate 100 networks. Figure 6.a shows the average performance as the size is varied and Figure 6.b shows the performance distribution of individual experiments. We observe that the AVG(PR/PZ) drops as the network size increases. This can be explained by the fact that as the size increases, the nodes that have geographic information are distributed more evenly and each node has higher probability to be in the convex-hull of several beacons. In our experiments, even for ranging errors of 5% and 10%, the first-class errors occur with a small possibility (<5%) and in all those cases the deviation of *PR*s from their actual positions are smaller than 1/20 of the communication range.




**(a) Average performance**




**(b) Distribution of performance**

**Figure 6 Performance of PRDS.**

### 3. Effects of Density

The network density is represented by average out-degree per node in our experiments. Figure 7 shows how this density affects the overall performance of PRD/DSS. The AVG(PR/PZ) drops steeply until density is approximately 7, where its value is around 1.4. After this point, it declines slowly and eventually converges slowly to 0 over a long range of density values. Clearly, the behavior of AVG(PR/PZ) follows a Zipfian distribution, a behavior common to dynamic complex systems.

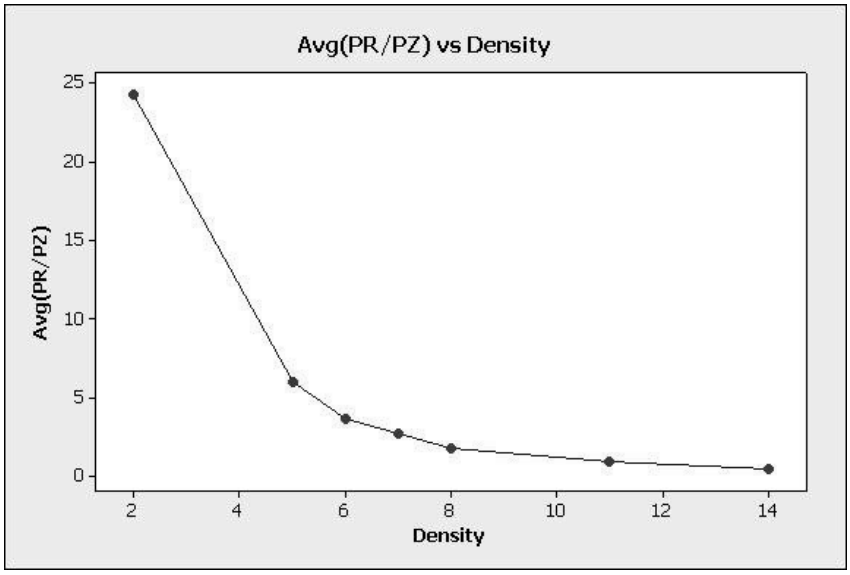



**Figure 7 Performance with varying density**.

## 4. Comparison

In Figure 8, the PRD algorithm is compared with the bounding box approach in [3] with respect to second-class errors (Avg(PR/PZ)). Our approach is clearly superior to [3], i.e., the average size of *PR* is significantly smaller (about 1/3 to 1/4) than the bounding box calculated in [3], on the same network conditions.

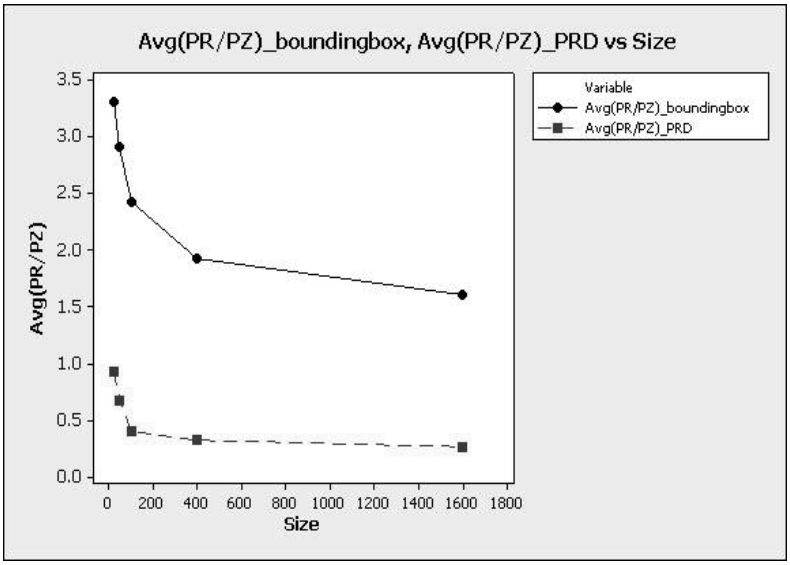



**Figure 8 Localization performance comparison.**

## I. CONCLUSION

We introduced an energy and bandwidth efficient localization and indexing scheme in ad-hoc sensor network. This will enable the design of sophisticated wireless sensor databases. We showed that with low message and time cost per node, the indexing structure is able to efficiently store and query the data in the network.

# III. SCALABLE SELF-CONFIGURING INTEGRATION OF LOCALIZATION AND INDEXING IN WIRELESS AD-HOC SENSOR NETWORKS

## A. Introduction

This section is an extension of the research in Chapter II **(**"PRDS"**)** which uses imprecise yet simple localization approach to discover zone assignment. A localization approach is applicable for indexing if it can successfully localize a sensor into a zone, and zone partitioning decisions are subsequently made locally at each sensor. The satisfaction of this criterion enables a scalable peer-to-peer emergent distributed space partitioning method. The relaxation of precision to the granularity of a zone speeds up localization considerably, and in turn query latency is improved and power consumption is reduced. With an index structure capable of adapting to topology changes, our experiments show that PRDS can usually perform quick location discovery and correct zone assignment with an initial core of at least 15**%** to 20**%** location-aware sensors.

Our two-phase adaptive algorithm, called "adaptive PRDS" in this Chapter, provides superior solutions with less than the percentage of location-aware sensors allowed by simple PRDS, a simplified version of PRDS, at the cost of a small increase in latency. When a moving sensor is able to determine its zone locally using the simple PRDS, zone partitioning and assignment is accomplished very quickly thereafter. On the other hand, when zone localization is ambiguous using the simple PRDS, location is pinpointed based on the sensor's interaction with neighboring nodes

acting as beacons. The research work in this chapter was published in [28] and covered under US Patent No. 7,840,353 [19].

## B. Related works

Vivaldi [29] and AFL [30] are two decentralized algorithms that use mass spring optimization method without reliance on any location-aware sensors. Vivaldi builds a network coordinate system to predict the communication latency between the hosts. Both approaches require significantly long time and large number of message exchanges to achieve convergence. In Vivaldi, the convergence takes at least 100 seconds [29].

## C. Preliminaries

### 1. Assumptions

Before describing our scheme, we shall make the following model assumptions: (i) the wireless sensor network is connected all the time regardless of sensor mobility and power shift. This is necessary for the mapping between data and location, and for the validity of query processing; (ii) a small fraction of sensors are equipped with GPS or pre-measured location information; and finally, (iii) sensors without GPS or other positioning capabilities are equipped with ranging devices to estimate their distance to other sensors within its communication range. Each sensor is able to share its identifier and position information with its neighbors. But it discloses this information to its neighbors only when specifically requested. In the sequel, sensor and node are used interchangeably if there is no ambiguity.

## 2. Definitions

*Physical space* refers to the geographic area encompassing the network of sensors. *Data or Event space* is the domain of sensor data/events streams generated and gathered within the network. If there is no ambiguity, we shall refer to event space and data space interchangeably. *Zone space* is the set of partitions formed as a result of partitioning the physical space. Each partition is called a zone. A zone is managed by exactly one sensor. Let *f* be a mapping from the data space to the physical space, and *g* a mapping from the physical space to the zone space. Given a tuple X in the data space, the composition of mappings $g(f(X))$ determines the sensor (zone) responsible for its management. The specifics of the mappings used in our scheme are given in Chapter II.

*Sensing range* is defined as the radius within which sensors can sense events occurring in the field. *Communication range* is defined as the maximum communication distance between two sensors. Two sensors are called *neighbors* if they are within each other's communication range. As in most literature, both ranges are viewed as circles centered at the sensor. *Sensing or communication coverage* of a sensor refers to the space within its sensing or communication range. *Data coverage*, on the other hand, describes the range of events managed by a sensor. Since each event is mapped to a physical location (or key) that basically gives its storage location, we define data coverage as the range of keys covered by the zone assigned to a sensor. The sensor identifier, which is also the zone identifier, completely characterizes the data range.

Each sensor's data coverage is determined based on the data partitioning approach. In our approach, the zone space is obtained by dividing the whole data space into rectangular zones, each assigned to a single sensor. A sensor stores all the events mapped to its assigned zone. There are several justifications for selecting rectangular zones instead of circular zones. First, the mapping of circular sensor range to rectangular data zone simplifies space partitioning. If the data space is partitioned into circular zones, the data zones will have to overlap with each other to ensure data coverage of all events. The overlapping of data zones creates ambiguity as to where an event should be stored. Moreover, the computation to delineate a circular zone managed by a sensor is much more complex than that of a rectangular zone.

The *position region* (PR) of a node n, denoted PR(n), is defined as a rectangle enclosing the likely location of n. Since the exact positions of sensors may not be known, their positions are approximated by their current position regions, the distance is obtained by ranging, which measures the distance between two nodes within their communication range. The PRDS (Position Region Data-Space partitioning) algorithm performs the dual action of deciding PR(n) and zone assignment to n by selecting one of n's neighbors to have its zone split, as covered in Chapter II.

## D. Algorithms

We first describe a condensed and slightly different version of simple PRDS algorithm in Chapter II. Then we introduce the two-phase adaptive-PRDS algorithm.

### 1. The simple PRDS algorithm

The simple PRDS algorithm proceeds as follows: given a new node, say *n*, unaware of its position, its position region PR is first initialized to be the whole space. The ranging distance to each of its neighbors enables PRDS to determine the rectangles bounding both it and the neighbors' position regions. The intersection of all these bounding rectangles becomes the new PR for node *n*.

Assuming *n*'s neighbors have stable position regions, the newly updated position region will also be stable. Otherwise it will itself be transient and further refinements will be necessary. The availability of location-aware sensors speeds up convergence to a stable position region. Clearly, frequent mobility of sensors slows downs convergence. A stable position is reached when no further refinement is possible.

After calculating PR(n), one of *n*'s neighbor is selected to have its zone split and one of the partitions assigned to *n*. The simple PRDS algorithm determines this neighbor and then performs the appropriate data space partitioning. The pseudo-code of simple PRDS algorithm shows how it performs the actions described above. The local data space partitioning phase begins when a stable position region is obtained. At this time, one of *n*'s neighbor is selected to have its zone split into two subzones,

one of which is assigned to *n* while the other remains at the neighbor. The simple PRDS algorithm is responsible for both selection and splitting. It selects the neighbor whose zone completely encloses or has the largest overlapping area with *n*'s position region PR. The message cost of simple PRDS algorith is *O*(*deg* * *c*), where *deg* is the sensor's out degree and *c* is a constant.

*PRDS(n)*

*//Steps 1 to 4 calculates position region PR(n) for node n*
*1. If node n is aware of its own position (px, py), PR(n) is trivially [px, px) × [py, py); EXIT.*
*2. Initialize PR(n) to the whole physical space [0,m) × [0, n)*
*3. For each neighbor ni of n, find PR(n, ni) and intersect it with the whole physical space:*
*PR(n, ni) = [PLx(ni) - d(n, ni), PUx(ni) + d(n, ni)) × [PLy(ni) - d(n, ni), PUy(ni) + d(n, ni)) ⋂* $[0, m) \times [0, n)$
*4. PR(n) =* $\bigcap_{i=1}^{\deg(n)} PR(n, ni)$
*5. For each neighbor ni of n, PRC(n, ni) = PR(n) ^ PZ(ni)*
*Select the ni with largest PRC(n, ni) as the splitting partner*
*6. Split ni's zone Z(I) until two zones Z(I1) and Z(I2) intersecting PR(n) and PR(ni) are obtained. Assign n and ni the enclosing zones Z(I1) and Z(I2), respectively.*

## 2. The adaptive PRDS algorithm

The simple PRDS algorithm correctly and efficiently localizes sensors and partitions the data space given an initial core of at least 15% to 20% location-aware sensors. Its performance degrades drastically when the size of the initial core falls below 15%. The PR(n) obtained by the simple PRDS may not be sufficiently refined to narrow selection to a single neighbor. The neighbor with the largest overlapping area with PR(n) may not be the correct one, as its zone may not cover node *n*. This imprecision may in turn cause incorrect zone assignment. This phenomenon is analyzed in more details in the simulation section.

The adaptive scheme can handle situations where the size of the initial core is below 15%. It consists of two phases. In **phase I**, each node *n* runs a variation of the simple PRDS algorithm to determine its position region PR(n) in a similar fashion as in the original simple PRDS algorithm. If PR(n) is contained in one its neighbors' zones, a split of this zone ensues and the algorithm terminates. On the other hand, if no such neighbor exists, instead of picking the neighbor whose zone has the largest overlapping as in simple PRDS, phase II is triggered. Node *n* sets its initial position to be the center point of PR(n) obtained in phase I, and then its actual position is refined by iteratively interacting with some of its neighboring nodes acting as beacons. The resulting position is mapped to an existing zone, which then gets partitioned such that each sensor manages the data associated with its mapped zone.

Next we discuss in detail the refinement method. Centralized or flooding-based location refinement algorithms in ad-hoc networks do not scale and are inefficient. Our distributed re-adjustment algorithm is driven by a classic force-based relaxation method, called mass-spring optimization, that locally minimizes the least square errors to refine the positions of sensors unassigned to zones, as discussed and applied in [30] and [29]. The distributed version relies only on local interactions with neighboring sensors, as in the simple PRDS algorithm. This ensures that the scalability and message efficiency achieved in phase I are also preserved in phase II.

In **phase II**, each sensor has an "*assigned*" flag to indicate zone assignment. This flag is set to 1 for all nodes successfully localized and assigned a zone; otherwise its value remains 0. Nodes whose "*assigned*" flags remain zero after a timeout will

proceed with phase II. It resets its location *Pn* as the center of its *PR*(*n*) obtained in phase I and broadcasts a phase-II message to its neighbors. Its neighbors respond with their own estimated position $P_i^n$ calculated from their position regions. The node then use both *Pn* and $P_i^n$ as well as the ranging distance *D*(*n*, *ni*) to pinpoint its location.

Suppose the direction from *Pn* and *Pi n* is represented by unit vector $\overrightarrow{d(n, ni)}$. The force $\overrightarrow{F(n, ni)}$ in the direction of *d*(*n*, *ni*) is then given by

$$\overrightarrow{F(n, ni)} = \overrightarrow{d(n;\ ni)} * (\mathrm{Pn} - \mathrm{P_n^i} - \mathrm{D(n, ni)}) \quad (1)$$

The position of node *n* is adjusted to a weighted displacement drawn in the direction of force *F*(*n*, *ni*). The new estimated position of *n* after the adjustment is:

$$Pn' = Pn + w * \overrightarrow{F(n, ni)} \quad (2)$$

The weight *w* is often set empirically in [30] as $1/2m_n$, and $m_n$ is the degree of node *n*. We set *w* to be between $1/2m_n$ and $1/m_n$, which by our experiments is proven to converge faster to the actual position than other possible values.

The convergence of the position calculated from phase II is fast. One reason is the preliminary position calculated from partial result in phase I is likely to be close to the actual position. The following shows the pseudo-code for the adaptive PRDS.

***AdaptivePRDS(****Pn****)***

*Phase I (Simple PRDS variation)*

*//Phase II begins, assigned is set to 1 if the zone has been assigned*
*while (not assigned) for each assigned neighbor ni*
*w = d(n, ni)/2m*$_n$

*//Compute unit vector between Pn and Pi n and determine the force in this direction*

$\overrightarrow{d(n,ni)}$=(Pn-$P_i^n$)/||Pn-$P_i^n$||

$\overrightarrow{F(n,ni)} = \overrightarrow{d(n;\ ni)} * (Pn - P_n^i - D(n,ni))$

*//Adjust the position*

*Pn' = Pn + w ** $\overrightarrow{F(n,ni)}$

*if timeout or (adjustment < $\varepsilon$)*

*assigned = 1g*

*K(ni)=identifier of ni whose zone contains Pn,*

*Z1=K(ni) appends 0 and Z2=K(ni) appends "1",*

*assign Z1 and Z2 to n and ni.*

---

This adaptive approach is capable of running in both the situations requiring extensive localization like initialization and those requiring simple localization like when a new node joins the network. In the situation of an initialization, because all zones are unassigned, it is possible that quite a few nodes in the network cannot decide its zone by phase I, posing the need for a further refinement in phase II. On the other hand, in the case when a "joining" node wants to calculate its position region, and most of its surrounding nodes are settled with stable PRs, the assignment is often completed in phase I without going into phase II.

The complexity of phase I has been discussed previously. The complexity of phase II is also bounded by O(deg * M) where M is the maximum number of adjustments allowed in a timeout period. However, phase II is often ended before M when latest adjustment is smaller than a predefined $\varepsilon$. It is also worth noting that when a large portion of nodes fail or dysfunction concomitantly and location determination is demanded without delay, our two-step adaptive PRDS approach can act without requiring an initialization step. It adapts seamlessly to situations that require strenuous localization or to those where only simple location discovery of a node is necessary without reliance on outside control.

## E. Experiments

The network of sensors is generated as a random graph with uniform node distribution. We tested on N varying from 50 to 3200. In our experiments, we define a variable parameter *f* to represent the ratio of location aware sensors over the total number of sensors.

We have conducted analysis of different combinations of levels of available geographic information and sensor movement patterns. In each, both first-class errors and second-class errors were investigated. In our experiments first-class errors occur very rarely (less than 3%) even with noise bound set to 10%.

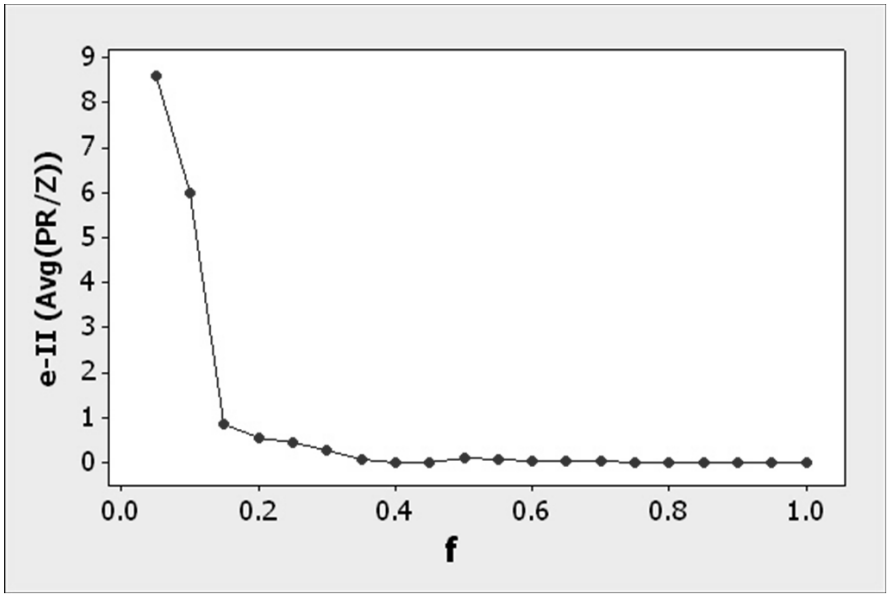



**Figure 9 Second-class error (granularity) measured by avg(PR/PZ) with varying f in a 100-node network.**

Figure 9 plots the second-class errors made by simple PRDS with different *f* values. In this graph the noise bound is set to be 10%. The plot tells that when f is lower than 12%, on average the size of PR becomes larger than the size of zone. We found that when Avg(PR/Z) is larger than 1, the probability of choosing the correct splitting partner is significantly lowered. The cause is that larger PR results in overlapping with multiple zones and thus ambiguity arises. When PR is larger than the average zone size, it is unlikely that it will be enclosed by a neighboring zone and thus will need to choose its splitting partner with probability. The plot tends to smooth out and converges to 0 after *f* is beyond 15%. This shows that with more location information in the network, simple PRDS alone has sufficient granularity and precision to perform the localization and data space partitioning.

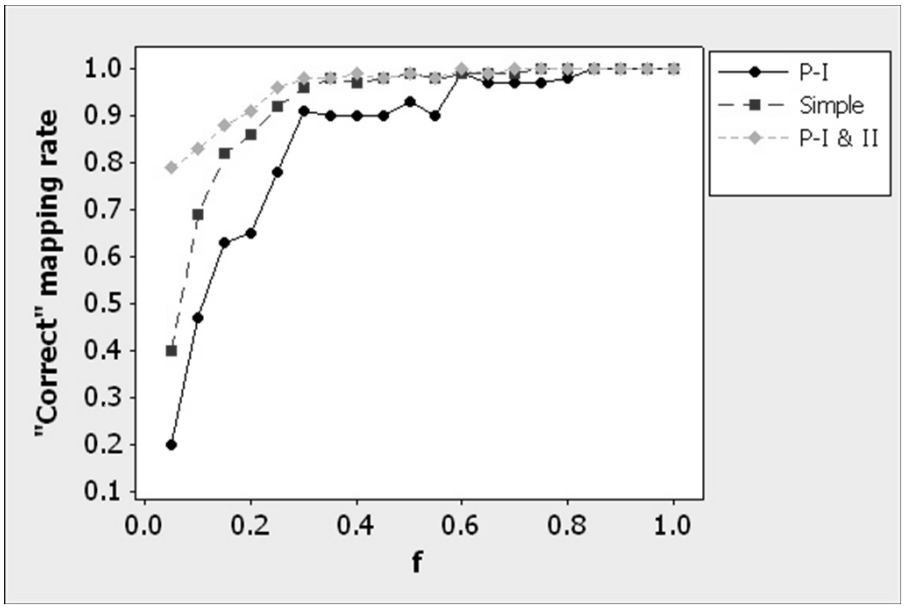



**Figure 10 Comparisons of different versions of PRDS in zone mapping ”correctness” with varying f in a 100-node network.**

Figure 10 plots the impact of different versions of PRDS on the ratio of “correct” zone assignments to nodes. If the chosen splitting sensor (zone) is the same one as the sensor (zone) in which the node is physical located, we say this assignment is “correct”. Maintaining “correctness” ensures the proper order of the space partitioning. The ”correctness” ratio is defined as ratio of the number of ”correct” assignments divided by the total number of assignments. All three plots converge to 1, because with sufficient location information, the three approaches perform basically the same actions that correctly assign the zones.

Now let us examine the three approaches separately. The first plot shows Phase-I (denoted as P-I) of adaptive PRDS when running alone. It is a variation of simple PRDS algorithm but does not make decision when the PR has overlapping with multiple zones, as does in simple PRDS. So it has lower ”correctness” rate than simple PRDS because of this.

The simple PRDS, which employs the policy of mapping the node to the zone with the largest overlapping when multiple overlapping occurs, has better ”correctness” compared to the former, which is achieved by occasions when it does ”correct” zone ”guessing”. However, when f decreases, the ratio of simple PRDS drops significantly. Especially, when f decreases to around 5% the ”correctness” ratio can drop to less than 50%. Clearly, this performance degradation does not suffice to maintain a properly ordered zone space.

The adaptive PRDS with both Phase-I and II achieves far better ”correctness” rate than the other two approaches when f is very low. It shows that the resulting

refined position is a closer approximation of the node's actual location than the PRs obtained from the other two approaches.

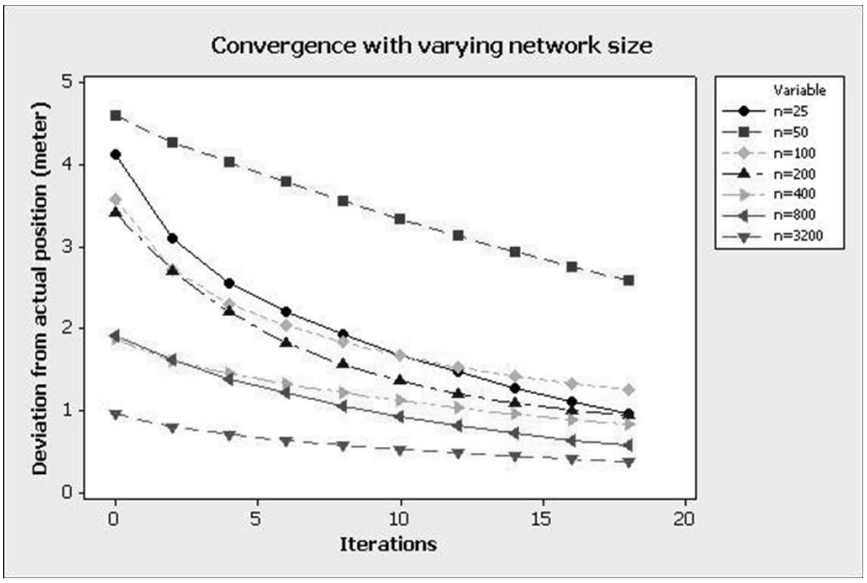



**Figure 11 Adaptive PRDS performance: convergence by iterations with varying network sizes**

In Figure 11 we evaluate the convergence of estimated position to actual position in networks of different sizes in adaptive PRDS. The *f* is set to 0.1, which means only 10% of sensors have geographic information. The weight *w* of adjustment caused by forces is set as $1=2m_n$ where $m_n$ is the out-degree of that node. This means that each adjustment is *w* times the force in distance in that force's direction. We observe that in around 10 iterations the plots start to smooth out and within 20 iterations all calculations converge to a relatively small deviation from the actual positions. The plots become closer to the actual positions when the network size

grows. This is because with such a small f (10 % here), smaller networks tend to have too little source of geographic information. For example, a network of 25 nodes only has 2 location-aware nodes. As network size grows larger, more location-aware nodes are available and tend to distribute more evenly throughout the network, which makes the calculation converge faster to the actual positions.

We have also conducted experiments to study relationships between localization granularity and network size or density. The result shows the granularity decreases with the increase of network size or density and follows a “Zipfianlike” distribution, a behavior common to dynamic complex systems.

# IV. DYNAMICALLY SELF-ORGANIZING SENSORS AS VIRTUAL IN-NETWORK AGGREGATORS AND QUERY PROCESSORS

## A. Introduction

The practical requirements and constraints under which query processing in WSNs has to be performed are best illustrated by revisiting the realistic application scenario described in the introduction: considers a modern health care facility where vital parameters of patients such as pulse, blood pressure, and body temperature are monitored through sensors attached to bodies of patients. As patients physically move in the health care facility, sensors form a dynamically organized sensor network. Sensors can also track the daily activities of patients and may also be deployed in rooms to collect environmental conditions at the health care facility such as humidity, noise level, and air temperature. All measured data are recorded as events, which are then stored within the network.

Sensor networks deployed in a such a scenario enable the continuous monitoring of patients without actually constraining them physically, while offering the possibility of immediate action in case of an emergency. Healthcare staff may simply be interested in finding out a patient's location, or a doctor may request "the medical records of all cancer patients whose body temperature is above 98 degrees today" or "the maximum and average body temperatures for each resident in the West Wing." A wireless mobile sensor network with database capabilities will transparently process such queries and return the answers regardless of patient's location or availability. This can significantly reduce the healthcare costs as well as

benefit the patients. With data being gathered, stored and aggregated in-network, regular reports could be sent to staff members automatically. Alarm notifications in emergency situations, such as a patient is experiencing a sudden heart attack, can be sent without delay, thus increasing survival and recovery chances.

From a data processing perspective, such scenarios require efficient data management and query processing techniques to support point, range and aggregate queries. Power and execution time efficiency as well as scalability of the query processing in terms of network size **(**the number of sensors**)** and data volume have to be guaranteed as well. Typical queries to be expected in a health care scenario include **(**such queries are also representative for many other application domains and serve as good overall test cases**)**:

- *SELECT pulse-rate FROM pulse-sensors WHERE resident-id = 998 (point query)*
- *SELECT resident-id FROM complex-readings WHERE systolic > 135 and body temperature IN (98, 102) (range query)*
- *SELECT COUNT(*), AVG(pulse-rate) FROM complex-readings WHERE diastolic > 100 (transient aggregate query)*
- *SELECT COUNT(*), AVG(pulse-rate) FROM complex-readings WHERE diastolic > 100 DURATION 1 week EVERY 30 minutes (continuous aggregate query)*

Aggregation operations in traditional database approaches are generally performed at the back-end at one or a few centralized servers after collecting all data from the network. This communication-intensive approach is not feasible in large-scale, data-intensive sensor networks due to power constraints of sensors. In-network aggregation [31] [32] [33] and group-aware network configuration methods [34], which calculate partial results along intermediate nodes on query routing paths,

perform much better in such settings and can significantly decrease energy consumption of sensors by reducing the communication and computation costs.

When an aggregate query is issued in TinyDB [31] [35] or Cougar [33], a routing tree rooted at the query issuing node is constructed to perform in-network aggregation along the tree. According to the aggregation mechanism of TAG [31], which is implemented on top of TinyDB, a tree is constructed dynamically by flooding the network from the source node. A different tree is constructed each time a query is issued from a different source node. Clearly, these aggregation processing mechanisms suffer from constant flooding of the network, when queries are issued by different nodes. Also, since each sensor node stores data within the full range, searching for records with a specific value or value range requires exhaustive search of the network.

The infrastructure in this chapter addresses these drawbacks and organizes the physical sensor network by partitioning the data space into subspaces and assigning each subspace to a sensor, which then takes over the responsibility of managing data belonging to this subspace. For routing purposes a simple overlay structure called AggIndex tree is constructed. The name AggIndex simply means "aggregate all the indexes." Its construction is based on the data space partitioning among sensor nodes. An AggIndex tree is shared by all aggregate queries. Updates to the AggIndex tree and to data space partitions, which may occur due to sensors going online/offline or moving, are always local, i.e., only affect a few nodes. The AggIndex tree supports aggregation along tree paths while concentrating query searches to affected subareas in the network, thereby reducing energy consumption and latency. It also facilitates

implementation of query aggregation optimization techniques such as query aggregation to further improve the performance of query processing. Analysis and experimental results presented show that our approach significantly reduces query routing cost and latency compared to other approaches **(**such as TAG, Gossip-based aggregation, centralized solutions etc.**)**, without incurring significant overhead for constructing and maintaining the proposed AggIndex overlay structure. Part of work in this chapter was also included in US Patent No. 7,840,353 [19].

## B. Related Works

### 1. Aggregation Related Literature

Aggregation has been studied extensively in the database community [36]. A significant milestone over the years is the work by [31], which proposes a generic framework, called TAG, to support aggregate in-network queries by constructing spanning trees rooted at predefined base-stations **(**"sinks"**)**. Query evaluation is done by first flooding a query from the requesting node into the network and constructing an aggregation spanning tree, and then collecting intermediate results and propagating them back along the tree by child-parent messages. A node may select more than one parent at a higher level, in which case the intermediate results are sent to each of the parents.

Spanning trees are also used in Cougar [32] and [33]. The TiNA [34] framework also exploits pre-existing aggregation schemes such as TAG, to reduce energy consumption in in-network aggregation.

[37] proposed a gossip-style protocol to compute aggregations in P2P networks. Each node assigns a non-negative probability to its neighbors with the sum of all probabilities being equal to 1. Then in the next round it sends each neighbor its share of the results according to the probability assigned. However, convergence of gossip-style protocols in general is very slow, as shown in [38] it requires at least $O(n^{1.5}\log n)$ transmissions even for optimized gossip protocols.

A large body of work has addressed the robustness and scalability of aggregates' computation using the concept of sketches [36] [39] [40] [41]. Sketches are created to compute approximate duplicate-sensitive aggregates, such as COUNT or AVERAGE, across faulty sensors. The basic idea is to use a hash function to map the value of a data tuple into the appropriate bit of the sketch. Query computation consists in first flooding the query across the network and then each node: (i) does the computation at its level and (ii) constructs its own local sketch before broadcasting it to the next higher level. These two latter steps are repeated in each iteration until the sink gets the final result. [42] introduced synopsis diffusion, a concept similar to sketches.

Some recent approaches focused on approximate query processing and energy saving by using methods like sampling and bit map estimation. For example, to overcome adversary interferences in the network, [43] proposed a tree sampling algorithm for approximate aggregation. Also, [44] proposed a robust algorithm for aggregation against high link loss. In addition, [45] designed a two-phase approximation scheme including (i) constructing a spanning tree from the sink node by broadcasting through the network; and (ii) dividing the data range of each sensing attribute into k sections and having each sensor maintain a k-bit map accordingly. Each parent node will then combine the bit maps of its children into its own bit map and so on.

Some other recent approaches [46] [47] did not use an indexing scheme but instead opt for a market driven supply-demand approach for delivering reports. These approaches deal with a different aspect of query processing and information delivery,

and therefore may be able to be used in conjunction with our indexing solution in a mobile setting.

## 2. Data Scheme Related Literature

Our goal in data management is to avoid excessive search or updating costs to query processing and routing, and perform range queries more efficiently as well as enabling smooth adaptation to topology changes in mobile contexts. We address this goal through the incorporation of a multidimensional indexing scheme for the sensor nodes.

Multidimensional indexing structures have been extensively studied in the past twenty years. See Chapter II for a literature survey. In wireless sensor networks, sensors are distributed over the physical space as ad-hoc and self-organizing agents. An appropriate indexing structure must emerge from the interactions of sensors and be adaptive to the topology changes caused by sensors online/offline or being mobile. We developed localization-integrated indexing of ad-hoc sensor networks in Chapter II and III.

DIFS [48] is an indexing structure that sets up a multi-rooted quad-tree when partitioning the data space. Each node has $2^l$ parents depending on its level l in the tree. Each leaf node stores the full data range of an attribute. Each intermediate node stores a portion of its children's range. For a node, all its parents cover exactly its data range. Meanwhile, a parent covers all its children's data histograms. Thus, a node

closer to the root of the tree will have more limited data range but broader data histogram. Multi-resolution storage techniques are also exploited in DIMENSION [49] [50], which uses in-network wavelet-based summarization and progressive aging of summaries to support long-term querying in storage and communication-constrained networks. The goal is to enable highly efficient drill-down search over summaries and efficient use of network storage capacity through load-balancing and progressive aging of summaries.

A two-tier query aggregation scheme was proposed in [32]. It runs on a central query manager and allows several access points. As a result it is possible to optimize duplicate/overlapping queries in order to save overall energy consumption in the network. However, the query manager located at the base station can suffer from the usual performance degradation associated with centralized stations.

In [51], a two-tier data storage strategy for answering precision-constraint approximate queries is described. A high-precision version of data is kept at the sensor, and a low-precision version at the base station. Some queries may be answered at the base station if the required precision can be met by the data in the base station. A refreshment policy keeps the base stations updated with the latest sensor data. In this case, this updating policy may be very costly.

Some recent approaches focused on query optimization. A recent literature survey on data gathering and dissemination was presented in [52]. For example, [53] proposed an optimization technique over TAG. The technique, called Pocket Driven Trajectories (PDT), establishes a data collection path for each monitoring query based

on the spatial layout of selected nodes throughout a sampling period of location aggregation. It then uses a base station to compute the PDT after the sampling period has ended. The end result is a Minimum Spanning Tree (MST) at the base station, which will be used for directing queries.

Some other recent research focused on improving the routing layer in distributed sensor databases. For example, [54] designed a routing layer that enables a sensor to select between two different routing protocols with reference to query category and network density. A central authority such as a base station was used to decide which routing protocol should be selected.

Some other recent approaches focused on the security aspect of data management, another important topic in sensor databases. For example, [55] proposed a protocol that allows a sink to verify the result of the query and prevents attackers from gaining information. Additionally, several other protocols have been proposed for monitoring the state of the network including node failures, coverage and exposure bounds, energy supply depletion, and topology discovery. For example, [56] described a centralized probabilistic approach, where approximations are computed with probabilistic confidence intervals at the sensors. Using the summary maintained by the individual sensors, a statistical distribution was built hierarchically. In addition, [57] proposes an online aggregation interface, which permits users to observe the progress of their queries and control their execution on-the-fly. All these approaches may be complementary to our work.

## C. Preliminaries

In order to obtain optimal storage and search cost for data in a sensor network, the basic idea of indexing a sensor network database is to partition a d-dimensional data space into a finite number of subspaces and assign the subspaces to be managed by sensors in a way to enable efficient query processing.

### 1. Data Space Partition

**Definition 1** *Let R has be a relation with attributes $R_1$,$R_2$, … , $R_k$, where each $R_i$ takes its value from a bounded domain $D_i$. An index is to be built along a subset of attributes {$R_1$, …, $R_d$} of relation R. Each $R_i$, $1 \leq i \leq d$, is called an index attribute of R. A tuple of relation R is a record t= ($t_1$, …, $t_d$, …, $t_k$) $\in$ R. The data space is defined as the space enclosing all tuples of relation R.*

**Discussion** on index attributes: The choice of attributes to be indexed is generally application-dependent. The usual goals are efficiency and flexibility to handle complex ad-hoc queries. Application data are analyzed to select those attributes with wide ranges of possible values or those that are frequently specified in queries. For example, for location-aware applications, 2D or 3D location attributes are likely to be selected, whereas in environmental applications, weather data such as temperature and precipitation are more likely to be used as indexing [58].

**Definition 2** *To create a uniform index system that can be applied universally to any relation with arbitrary property domains, each index attribute value $R_i$ is normalized*

*to $b_i \in [0,1)$. The normalized value is called the mantissa of the corresponding attribute value. Thus, the index attribute set of R becomes $\{b_i \mid 1 \leq i \leq d, b_i \in [0,1)\}$, where each $b_i$ has a binary representation $b_{i,1} \ldots b_{i,max}$. We refer to relation R's data space thus normalized as relation R's normalized data space.*

**Example** Consider for example of a 2-dimensional use case where the two index attributes *D*1 and *D*2 are pulse rate (40 beats/minute – 160 beats/minute) and systolic blood pressure (70mm Hg – 230mm Hg). A tuple (130, 190) will be normalized to ((130, 40) / (160 - 40), (190 - 70) / (230 - 70)) = (90/120, 120/160) = (0.75, 0.75), which is represented in binary form by (.11, .11).

**Definition 3** *The key of a data tuple $t = (t_1, t_2, \ldots, t_d)$ is calculated by a perfect shuffle operation interweaving cyclically the bits of each $t_i$'s binary representation $t_{i,1}, \ldots t_{i,max}$:* $key(t) = t_{1,1}\, t_{2,1}, \ldots t_{d,1}\, t_{1,2} \ldots t_{d,max.}$

**Definition 4** *A data regions is a hyper-rectangular subspace within the data space. Each data region is uniquely represented by an identifier called r-id, which is a binary sequence $b = i_1 i_2 \ldots i_l$ with length l. l is also called the level of a data region. The collection of all data regions in the data space constitutes a data space partitioning.*

Note that all operations on keys or r-ids are performed on their corresponding binary representations. Unless specifically mentioned, we always refer to a data region by its r-id and we denote keys or r-ids by their binary presentations. For example, data region defined by r-id =1000 is referred to as data region 1000. Figure

1 illustrates the partitioning the data space into 9 data regions each denoted by their r-ids. It is trivially clear r-ids are unique in a given data space partitioning.

**Definition 5** *Let prefix() and max_prefix() denote two mappings on relation R's normalized data space. Given two arbitrary binary strings b and b' in the normalized data space, b' is said to be a prefix of b, b' = prefix(b), iff b' is a prefix binary substring of b. It is said to be a maximum prefix, b' = max_prefix(b), iff there does not exist a shorter substring b such that b' = prefix(b).*

**Lemma 1** Given R-ID = {r-id | data region r-id is in the data space partitioning}. If the range of mapping max_prefix is restricted to R-ID, instead of relation R's normalized data space, then a max_prefix is a function, i.e., given any arbitrary string b in relation R's normalized data space, there exists a unique r-id mapping in R-ID.

**Proof**: The proof is trivial. By construction, given a key of a tuple t, it is mapped to data region r-id = max_prefix(key) in the R-id tree. Tuple t is stored in the sensor handling data region r-id.

Consider again the health care use case, the key of tuple (130,190), calculated by interweaving cyclically the bits from (11, 11), is 1111, assuming l=4. Then the key 1111 is mapped to a data region by function max_prefix(), which, according to Figure 1, appears to be data region 1111.

**Definition 6** *Let b be the binary sequence of a data region. A split along the i-th dimension subdivides it into two data regions, which r-ids are obtained by appending*

*0 or 1 to b. A merge of two data regions b and b' in the R-id tree combines them into a single hyper-rectangular region. Thus, a merge is possible iff their bs differ only in their last bit. The b of the newly created larger data region is simply the largest common prefix of the two original data regions.*

For example, a merge of data region 1110 and 1111 produces data region 111. Refer to Figure 1 for an example.

More details of this approach for organizing an indexing structure can be found in [21].

## 2. The R-id tree

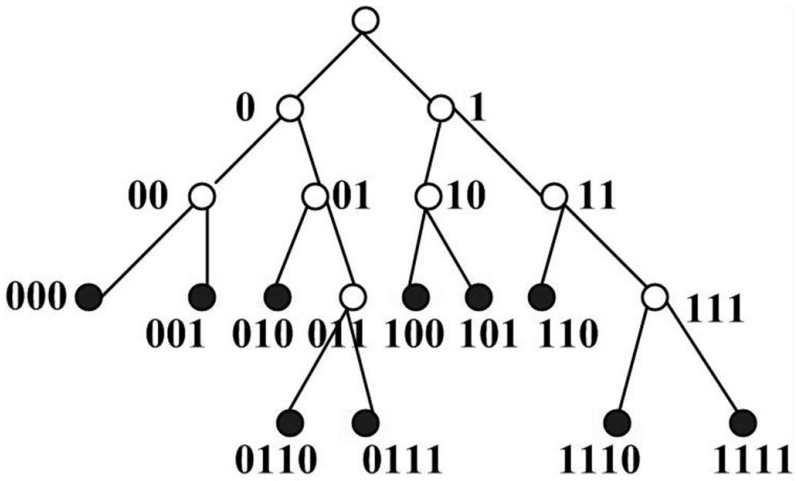



**Figure 12 R-id tree**

The tree shown in Figure 12, called the R-id tree, corresponds to a data space partition. Each of its leaf nodes represents a r-id of an existing data region, referred to as real identifiers. The intermediate nodes represent old r-ids of data regions that have

been split to obtain the current data space partitioning. Therefore, the intermediate nodes represent the history of data region splits. We referred to them as the virtual identifiers.

The recursive binary decomposition process results in a data space partitioning where data regions whose corresponding r-ids form the leaves of the tree. The leaf nodes are assigned to sensors. The intermediate nodes in the R-id tree do not represent actual data regions but they may be maintained to facilitate routing and query processing in the network. We will introduce our AggIndex tree scheme, which is based on the R-id tree concept but actively maintains the intermediate nodes of the tree.

In Figure 12, a split of data region 111 generates two data regions 1110 and 1111. Data region 111 becomes a parent node in the tree with children 1110 and 1111. The sensor holding 111 is now responsible for the management of data region 1110 while the incoming sensor handles data region 1111. Data region 111 is no more, but as will be discussed in the AggIndex tree scheme in the next chapter, the id will be kept at the sensor originally identified by 111 and becomes an intermediate node in the AggIndex tree structure.

### 3. Physical and sensor space

Unless there is ambiguity, we always use sensor and node interchangeably. In our approach to data organization in a distributed environment of sensors, each data

region is mapped to a unique sensor to enable collective and collaborative management of the data space. The mapping between the partitioned data space and the physical space of sensors is defined below:

**Definition 7** *A physical space is the geographic area covered by the network of sensors. It is partitioned into subspaces of sensors according to their locations. Let f be a mapping from the data space to the physical space, and g a mapping from the physical space to the data space partition. Given a tuple t in the data space, the composition of mappings* ***g(f(t))*** *determines the data region responsible for its management. The mapping between spaces is illustrated in Figure 2.*

When a sensor joins the network, it is assigned a r-id. It becomes responsible for the storage, maintenance, and query answering of those tuples mapped to the data region r-id. Let key be the key of t, then r-id = max_prefix(key). Unlike intrinsic properties such as permanently assigned MAC addresses or identification numbers, r-ids are dynamically assigned to sensors according to sensor locations and may be updated by sensor movements or mode switches.

**Definition 8** *The physical space keeps splitting into subspaces called zones until every sensor uniquely delineates a zone for itself. When a new sensor moves into a zone already occupied by a sensor r-id, the zone will be split between the two sensors. This also causes the split of data region r-id between the two sensors.*

Sensors become neighbors if they can communicate directly. Each sensor also maintains an up-to-date r-id list of all its neighbors. More details on the mapping of

physical space to data space and zones to data regions were exploited in Chapter II. Note that while the logical space, relation R's data space, defines all the possible sensed values, it does NOT guarantee that all event occurrences in the geographical space are sensed. Coverage of the geographical space depends on sensor placement, and therefore the space covered by the zones.

**Discussion 1** The mapping between data space and physical space preserves the locality property, as shown by Theorem 1. Alternative approaches such as W-Grid [59] maps data space to a virtual space where sensors are represented by virtual coordinates without any knowledge of sensor locations. However, two data tuples close in data space may be mapped to sensors close in virtual coordinate space but far apart in physical space. In effect, preservation of physical locality is not guaranteed there.

**Definition 9** *A mapping function h(t) is C-locality preserving if two data points are always mapped to locations within a constant C times their original distance in the data space.*

**Theorem 1** *The mapping function of g(f(t)), defined above, is C-locality preserving, and C =* $\frac{L}{min(Di)}$*, where Di is the data range for dimension i and L is the length of the physical field.*

**Proof**: Suppose two data tuples are separated by distance x in the data space. Their corresponding keys then have a distance y by function f. The physical distance between the two tuples is z by mapping function g. We prove that $z/x \leq C$.

Suppose the data space has k dimensions, among which d dimensions are selected to build the index. On average, the above two data tuples are mapped to within distance $2^{d*\left\lfloor log\frac{x}{\sqrt{k}*\min\{Di\}}\right\rfloor} * L, 1 \le i \le d$ in the physical space. The projection of x on each dimension in the data space is $\frac{x}{\sqrt{k}}$. It becomes $\frac{x}{\sqrt{k}*Di}$ after normalization.

On average, two data points separated by distance x will differ in the i-th dimension after $\left\lceil -log\frac{x}{\sqrt{k}*Di}\right\rceil$ splits in that dimension. Since the key is created by taking cyclically from each index dimension (d out of k dimensions), keys of the two data points will be differing starting from the d*$\left\lceil log\frac{\sqrt{k}*\min(Di)}{x}\right\rceil$ th digit, $1 \le i \le d$. Therefore, z is $2^{d*\left\lfloor log\frac{x}{\sqrt{k}*\min(Di)}\right\rfloor} * L, 1 \le i \le d$.

In the worse case, the two data points have the same coordinates in all other dimensions but one. Therefore, the keys start to be different at the $\left\lceil (-d) * log\frac{x}{Di}\right\rceil th$ digit. Because $\left\lceil (-d) * log\frac{x}{Di}\right\rceil \le \left\lceil (-d) * log\frac{x}{\min(Di)}\right\rceil, 1 \le i \le d$, then $\left\lceil (-d) * log\frac{x}{\min(Di)}\right\rceil$ is the first different digit in the keys of the two data points.

In the worse case, the distance $z \le 2^{d*log\frac{x}{\min\{Di\}}} * L = (\frac{x}{\min\{Di\}})^d * L, 1 \le i \le d$. Therefore, $z/x < \frac{(\frac{x}{\min\{Di\}})^d * L}{x} = \frac{L}{\min\{Di\}} * (\frac{x}{\min\{Di\}})^{d-1}$. It follows when x< min{Di}, z/x $< \frac{L}{\min\{Di\}} = C$. When x≥ min $\{Di\}$, z/x < L/x $\le \frac{L}{\min\{Di\}} = C$ (z < L by definition and x≥ min $\{Di\}$,). □

Theorems 2 through 5 will be presented in Section E (Analysis) of this Chapter following next Section D's description of query processing.

## D. Query Processing

### 1. The AggIndex tree

The AggIndex tree is a tree overlay based on R-id tree introduced earlier. However, unlike R-id tree where only the r-ids are maintained, here both the r-ids (as leaf nodes) and virtual ids (as internal nodes) are maintained at the sensors. Therefore, the AggIndex tree is a "real" tree structure connecting the sensors and the data regions in the network. In an AggIndex tree, each sensor node maintains one r-id: the real id, or r-id, which represents the actual data region currently assigned to the sensor. In addition, a sensor may be assigned some virtual ids (v-ids), which become internal nodes in the AggIndex tree. Those v-ids are old r-ids of those data regions that have since split. A sensor holding a v-id does not directly manage the data region corresponding to the v-id, but is in charge of query routing and processing of data mapped to that v-id. Figure 13 below illustrates the AggIndex tree corresponding to the R-id tree in Figure 12. For example, a sensor node with a r-id 0110 may also hold v-id 011, if at some point, data region 011 was managed by the same sensor and subsequently went through splits.

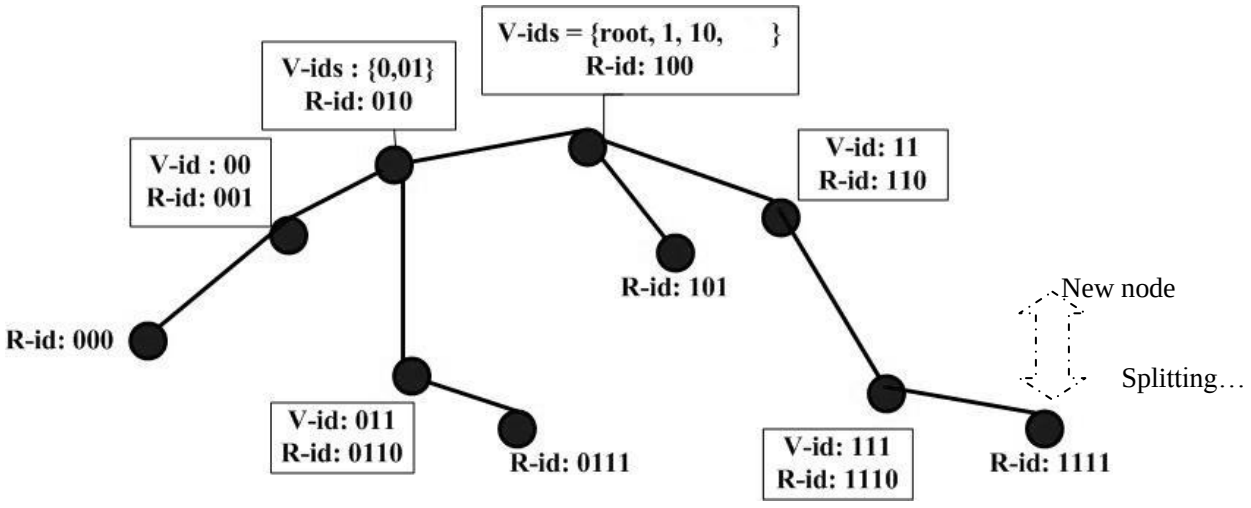



**Figure 13 AggIndex tree**

## 2. The history approach of constructing the AggIndex tree

In a static environment, where network topology and node reachability are stable, an AggIndex tree is constructed and maintained through the utilization of the space partitioning history. Updates to the AggIndex tree are localized. Each sensor node stores its r-id and a list of v-ids locally. Each r-id or v-id has a parent in the AggIndex tree.

Next is the corresponding pseudo-code for the update procedure in Figure 14.

**aggindexHistory** ***(oldrid, rid, updatetype)***

```
if (updatetype == "split")

    {rid = oldrid appends 1 or 0;
    rid.parent = oldrid;
    vidlist.add(oldrid);}

else if (updatetype == "merge")

        {rid = prefix of rid;
        rid.parent = oldrid.parent.parent;
         vidlist.delete(oldrid);}
```

**Figure 14 AggIndex tree: history update pseudo-code**

Two types of updates may occur, a "merge" or a "split." In the pseudo-code above, old r-id refers to the id of the sensor before the merge or split. R-id is the new id after the merge or split. When a split takes place, r-id is split into two by appending 0 or 1 to the end of the r-id, and the parent of the new r-id is set to the old r-id. The old r-id is also added to the current node's v-id list, thus becoming a virtual id, an internal node in the AggIndex tree. When a merge takes place, two data regions

merge into one. The r-id is set to the new data region resulting from the merge by taking the prefix of the two merging r-ids. The parent of the new r-id is set to the grandparent of the old-id. And old r-id is removed from the v-id list.

This process can be illustrated using Figure 13. Sensors are represented by blackened nodes and their ids are shown in the boxes surrounding the nodes. Node with r-id 100 is the root of the tree. It manages data region 100. In addition, it also maintains a v-id list, which includes the root, the intermediate ids resulted from splitting down to its r-id, namely, 1, 10. It has a child whose r-id is 010 (vids 0, 01 - vid 0 is a child of vid root, which is the node with r-id 100). It also has a few grandchildren whose r-ids are 110, 1110 etc. Similarly, node with r-id 010 has a child whose r-id is 001 and a grandchild whose r-id is 000. Node with r-id 000 manages data region 000. Its parent is the node with r-id 001 and v-id 00. Now let's see a scenario of a new sensor coming in. When a new sensor joins the network and interacts directly with a sensor (r-id 1111) in the network, data region 1111 is split into two data regions, 11110 and 11111. After the split, the first region, r-id 11110 is assigned to and managed by the new sensor. The assignment was based on the relative location of the two nodes. The second region, r-id 11111 remains at the current sensor, i.e., the node being split. Then, old r-id 1111 was added to the v-id list of the sensor being split and the new node becomes the child of the node being split. The split history is thus recorded in the AggIndex tree, which allows the maintenance of the internal nodes after many data region have been split or merged.

The height of an AggIndex tree is is $O(\log_2 N)$ on average. It is determined by the length of the longest r-id in the network, which is on average $O(\log_2 N)$. Thus,

history update incurs minimal additional cost on the average. It is very efficient for environments where sensors are stationary. On the other hand, when sensors are mobile, it may perform sub-optimally due to topology changes, which cause the historical information to be outdated and/or the nodes to be disconnected. The issues of mobility and topology changes are addressed below, where, in particular, a dynamic update method is presented. Note however that, in cases of skewed distributions, the height could increase significantly causing the performance to deteriorate.

## 3. Dynamic update for mobile environments

In a mobile environment, where topology is continuously changing, the maintenance of an AggIndex tree is more complicated. Consider the example in Figure 15, which shows an AggIndex tree set up in a physical area of sensors. Tree links from child to parent are represented by arrows in the graph. Each node is represented by an identifier consisting of r-id and a list of virtual identifiers v-ids. Let us assume that sensor 110, {11} moves out of the range of its parent node 100, {10, 1}. This causes the AggIndex tree to become disconnected, and thus, useless for properly routing the queries. Restoration of connectedness is thus paramount, but it must be done in a way that maintains its properties.

We designed a new algorithm to deal with the disconnection issue and the dynamic maintenance of the AggIndex tree. Whenever a sensor detects a disconnection with its parent node, the dynamic update method is triggered to readjust

the AggIndex tree. Each node keeps track of its neighbors in a neighbor-list, which contains all its neighbors' ids and the parent r-id. When a parent node cannot be located within a fixed period of time, a lost-parent message is broadcast in its neighborhood. The search for lost-parent is similar to pathDiscovery algorithm in routing. If the lost-parent is discovered through routing on existing AggIndex tree links, the AggIndex tree is adjusted accordingly. Otherwise, links not currently on the AggIndex tree are exploited to build a path to the parent. In the case where a parent cannot be located within the maximum allowed number of iterations, lost parent's parent (grandparent) in the AggIndex tree, and recursively thereafter, will temporarily serve as parent. The pseudo-code for the dynamic update procedure, dynamicUpdate, is given in Figure 16. The dynamic update method incurs a higher message cost than the history method, but updates are still done locally and thus the overhead is kept small.

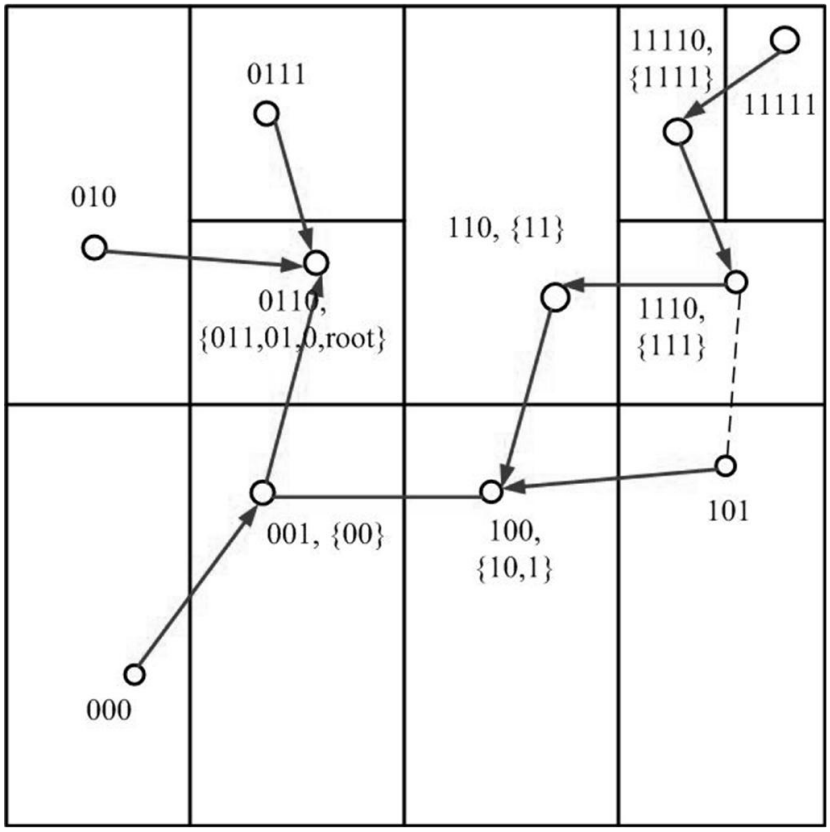



**Figure 15 Dynamic update of AggIndex tree in a physical network of sensors.**

**aggindexDynamic()**

---

if (*parentconnection* == *unvailable*)
{initiate message *reAdjustTree(rid, vidlist, lostparentid)* and send it to all children;}

While (*time* < *TTL*) {
*Upon receiving a* re-connect *message from node m:*
*parent* = *m*; Update *vidlist.*

*Upon receiving* reAdjustTree *message from node m:*
check if ((*parentlostid* appears any of its neighbors k) or (*parentlostid* belongs to any neighbor k's parent))
Yes: send *reconnect*(self) to m; *parent* = *k*; Update *vidlist*;

No: forward *reAdjustTree* message to its children attaching n's own information {*rid, vidlist*}
}
*parent* ← *oldparentid.parent*;

**Figure 16 Dynamic update algorithm.**

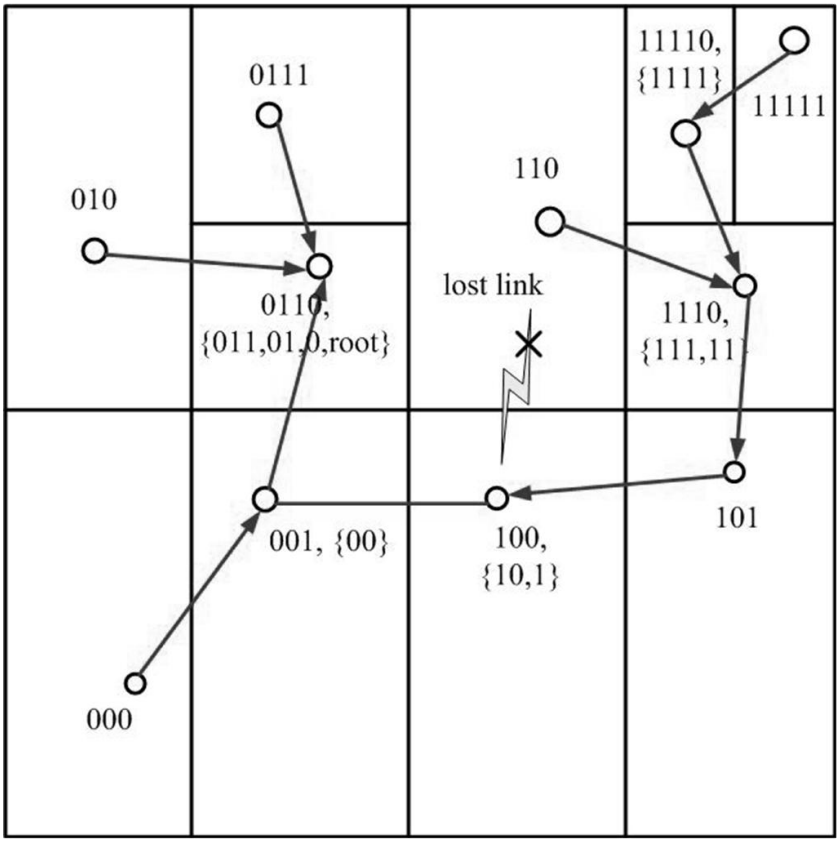



**Figure 17 Dynamic update of AggIndex in a physical network of sensors after readjustment caused by lost link.**

**Example** Consider the case in Figures 15 and 17, node 110, {11} (i.e., node 110) broadcasts a lostParent message including its own id: 110, {11} and lost parent's id: 100, {10,1} (i.e., node 100) to all its neighbors (in our case, node 11110 and 1110). Node 11110 does not have a connection to the lost parent, so it itself forwards the message to its own children, in this case 11111 only, but it appears that none are directly connected to the lost parent. However, one of 110's children, namely node 1110, is within communication range node 101, which itself is a child node of the lost parent 100. When 101 gets the forwarded message from 1110, the path to the lost parent 100 is rediscovered. Then, node 101 will send a findParent message back along the same path travelled by the lostParent message. Every node in the path will

mark the message sender as its new parent node. The v-id list will also be adjusted accordingly by removing obsolete v-ids. The updated AggIndex tree is shown in Figure 17.

### 4. Efficient in-network aggregation

Aggregate queries in general are very important in many applications. Processing them efficiently is particularly difficult in mobile sensor environments. We propose to show how the AggIndex tree can be utilized to efficiently answer aggregate queries. Processing a query consists of three main steps: (1) Query routing for directing a query to the vicinity of the sensors to be searched; (2) Local search and result aggregation; and (3) Result collection at the query sink node.

#### i. Definition and Assumptions

Once a query is generated, the first phase of query processing is to route the query to the vicinity of the sensors to be searched. The range of the sensors to be searched is determined by looking at the attributes and ranges specified in the query. Here the AggIndex structure organizes the data space, while preserving locality, into a set of disjoint data regions. This property is helpful in routing decisions and in reducing the number of messages. It is exploited here to design routing algorithms, and in the experiments to show the reduction in messages during query processing.

The general picture is the following. Before a query is routed, a query key is calculated. Then during query routing, the query key is mapped to a physical location or a region where the sensors are located. The query is then forwarded toward the physical location or region. A sensor within the location or region is selected to collect the result from other sensors. That sensor uses the established AggIndex structure to obtain aggregated information for the query, thus reducing the time and messages to be sent for query processing.

**Definition 10** *A query key, denoted qk, is a key calculated from a query q's specified range.*

For a point query, qk is calculated as the key of the target tuple by interweaving the bits of its normalized attribute values, following the method described in Section 3.1. For a query with a continuous range, qk is calculated as the common prefix of bounding values of the query range R. If a query has multiple disjoint ranges $R_1$, $R_2$, ..., $R_i$, because each range $R_i$ will be represented by a subtree in the AggIndex, a different $qk_j$ is computed for each disjoint range $R_j$.

The following is the pseudo-code for calculating query keys (qks), first for calculating point queries (the point to be searched was denoted by a d-dimensional vector searchvalue), followed by the procedure for calculating the continuous range queries (the range was denoted by [searchvalue_lower, searchvalue_upper]).

---

```
Procedure singlekey (searchvalue) {
        d is defined as the dimension of the range;
        define mantissadata [d];

        for (int i=0; i<d; i++) {
        mantissadata[i] = searchvalue[i]-datalowerbound[i])/(dataupperbound[i] -
        datalowerbound[i]);
        }

        for (int k=0; k<maxlength; k++) {
                for (int j=0; j<d;j++) {
                        binarykey appends ((int) (mantissadata[j]*2));
                        mantissadata[j] = mantissadata[j]*2- (int) (mantissadata[j]*2);
                }
        }
return binarykey;
}
```

```
Procedure rangekey(searchvalue_lower, searchvalue_upper) {

        lowerkey = singlekey(searchvalue_lower);
        upperkey = singlekey(searchvalue_upper);

        //binarykey is the max prefix of lowerkey and upperkey
        boolean same = true;

        for (int i=0; (i<d*maxlength)&&same; i++) {
                if (lowerkey(i)==upperkey(i)) binarykey(i) = lowerkey(i));
                else {same = false;}
        }

return binarykey;
}
```

A query q with key qk is routed toward all sensors whose r-ids are prefixes of query key qk, as explained next in the next subsection on routing.

Before describing our approach, we first establish some assumptions about the sensing and communication range of sensors.

a. For simplicity, the sensor network environment covers a bounded two-dimensional rectangular area as the extension to three dimensions is straightforward.

b. Every sensor has the same sensing range $r_s$. The sensing coverage of a sensor is a circle $(p,r_s)$ centered at the sensor location p with radius $r_s$. A physical space of sensors is sensing-covered if any point in the physical space is covered by at least one sensor.

c. The communication coverage of a sensor is a disk of radius $r_c$ centered at p, where communication range $r_c$ is the same for all sensors. Two sensors s1 and s2 can directly communicate if.f. distance $d(s1,s2) \leq r_c$. We assume the wireless sensor network is sensing-covered and connected all the time

regardless of sensor mobility and power shift, even for ad hoc network deployment. Recent research [60] on the impact of sensing coverage on greedy geographic routing shows when the range ratio $r_c/r_s$ satisfies a double range property, i.e., $r_c/r_s \leq 2$, a sensing-covered network is always connected. Since network connectivity is necessary for any routing algorithm to find a routing path, it is reasonable for us to assume the double-range property which always results in a connected network.

d. We do not deal with MAC layer protocols here. However, most MAC layer protocols developed for sensor networks such as S-MAC and Zigbee **(**802.15.4**)** can be readily incorporated in our application.

### ii. Routing Algorithm For Query Propagation

In order to propagate the query to the vicinity of the sensors to be searched, a routing method is needed. The routing method needs to take into consideration the physical connection and geographic capabilities in the network and reduce the traffic as much as possible. Geographic routing [61] [60] has been shown to be a suitable and efficient routing scheme for propagating the query to the result holding sensors because it is a shortest distance approach and no routing table needs to be maintained. Greedy forwarding (GF) [61] [60], is one form of geographic routing. It always forwards a packet to the neighbor that is closest in distance to the destination. Its low overhead makes its implementation easier in a resource constrained sensor network. However, GF suffers from the local minima problem, a situation where no neighbor exists that is closer to the destination than itself, and a recovery method is necessary

to recover from the local minima.

One well-known approach to recovery from local minima is geographic parameter routing (GPSR) [61], where packets are routed around the faces of a planar sub-graph of the network until a closer node to the destination is discovered. Planar graphs are graphs with no crossing edges. They are generated from the original network graph by removing edges. However, Relative Neighbor Graph and Gabriel Graph generated by GPSR are not good spanners of the original graph because the distance between two nodes separated by a few hops in the original graph might increase significantly after planarization [60]. The correctness of planarization, and thus the robustness of GPSR, is compromised by errors caused by inexact position information and incorrect removal of cross edges [62]. This makes the development of a more robust routing mechanism essential to compensate for the adverse effects on routing correctness due to imprecise location information and relatively large signal errors from mobile sensors, particularly with our localization-integrated approaches in the previous chapters that calculate imprecise location information.

In [61] [62] [60], it has been shown that, under the double coverage property, GF always find a routing path between nodes u and v within $\left\lfloor \frac{distance(u,v)}{rc-2rs} \right\rfloor + 1$ hops, and the stretch factor under GF is asymptotically $\frac{rc}{rc-2rs}$. Therefore, in dense networks, routing can be based solely on GF.

Our scheme uses GF to forward the query to the vicinity of sensors to be searched. When local minima situation arises, two possibilities are exploited: (1) if

the current sensor is connected to any node in the sub-tree (denoted by the v-id held at the sensor) of the query root in the AggIndex: If this is the case, the query can then be forwarded to that node in the sub-tree, and that neighbor will deliver the query to the query root (defined as follows) along its AggIndex tree path; or (2) if not connected to the subtree, then iterative Breath-First Search (iBFS) is triggered for route discovery. This would guarantee a more robust routing with better success rate without GPSR's reliance on correct planarization and precise location information.

**Definition 11** *The query root of a query q is defined as the sensor holding the v-id or r-id that is the same as the qk calculated from query q. Because each v-id and r-id is unique and nonoverlapping according to our indexing scheme, the match is 1-1. If no exact match is found, the sensor with r-id that is the max-prefix of the qk becomes the query root.*

Our routing algorithm is informally described next, followed by pseudo-code in Figure 18.

Upon receiving a point query, each node checks whether its current r-id is a prefix of qk. If it is the case, a query match may be found from searching the data stored at this node. If not, the node will forward q to the next hop selected the nextHop algorithm in Figure 9, which will select the next hop for the q to be forwarded to.

Upon receiving a range query, a node checks if it is the query root by comparing the list of ids it maintains with qk. If at any point in the query propagation phase the sensor encounters a neighbor node belonging to the subtree of the query key qk or is connected directly to such a node, the query pk is forwarded to that node. That node will then use the

AggIndex links it maintains to deliver the query to the query root. If it determines it is not the query root nor does it belong to the subtree of the query key qk, it will use nexthop to forward the query.

When forwarding a packet pk, a node n runs the nextHop algorithm to determine which neighbor is the next hop in pk's routing path. The Euclidean distance delta**(**li, pk.dest**)** between each neighbor ni's location li and pk's destination dest is calculated. The neighbor, which has the shortest Euclidean distance and is closer than n to the packet destination, is chosen as the next hop.

If such a node or a sub-tree node cannot be found among n's immediate neighbors and none of its neighbors is part of the subtree of qk, it will start a discovery phase which uses iterative breadth-first-search **(**iBFS**)** technique to recover from the local minima. This is done by setting the iteration number to be 1 and broadcasting an ibDiscovery message including pk's destination information, its own information **(**such as location and id**)**, and iteration number to its immediate neighbors. See Figure 18 for the algorithm nexthop. If in one TTL none of its neighbors reply with an ibResponse message indicating the discovery of a closer node, n will increment its iteration number by one, and wait for another TTL until an ibResponse message is received from one of its neighbors. Here the TTL is set to the upper bound for a round trip duration between two neighboring nodes. A maximum number of TTLs can be set to avoid deadlock.

The process of an iBFS is the following: On receiving an ibDiscovery message from n′, a node n″ runs the pathDiscovery algorithm to determine whether a neighbor is closer to dest than n. If a closer node is discovered along the path, the ibDiscovery

process terminates and an ibResponse including its own information will be sent to n'. Otherwise a similar waiting period will be issued for the node to hear from farther nodes.

The supplemental use of AggIndex structure in routing provides better stability and less chance of falling into local maxima than simply relying on GF. As a tradeoff it may slightly increase the message cost in routing. In theoretical analysis we compare the two approaches, which will appear in the next section immediately following the analysis of the AggIndex topology, in theorems 4 and 5.

To facilitate processing a query during routing, some nodes along the routing path may preprocess some of the query results before the query gets to the query root if one of the two following applies: either (1) its r-id is a prefix of qk; or (2) qk is a prefix of its r-id. Either condition is possible, because a query may cover a broad range of data so that it may have a qk shorter than its r-id (see Figure 18 nexthop algorithm). Conversely, it may also cover only a narrow range of data and has a query key longer than a sensor's r-id. In either case satisfying sensors may hold potential results, so they will be searched for tuples that match the specified query range.

If no node is found matching a qk, i.e., the query is directed toward a hole in the data space not covered by any sensor, the query will return an empty result.

**queryrouting(*n*, *q*)**

//The query q here has only one qk
//when a query has disjoint ranges r1, ..., ri, a query qk(i) will be generated for each disjoint range ri, and the query splits into q1, ..., qi, each with a qk(i). Each qi will be routed separately.
if (*q.type* = "multi")
$q.qk = multikey$ = common prefix of q's enclosing values: $(lowerrange, upperrange)$
else if (*q.type* = "single") *q.qk* = $singlekey(q)$
next = $nexthop(n, q)$, forward *q* to next.
if ($n.vid == q.qk$ or $n.rid == q.qk$) or ($n.rid = max_prefix(q.qk)$) $q.queryroot = n$;
start query processing aggProc(q);
if (($n.rid$ is a prefix of $q.qk$ or *q.qk* is a prefix of $n.r - id$) and *q.type* = "multi")
start query processing aggProc(q);
search in $n$'s storage for tuples within *(lowerrange, upperrange)* retrieve all matching tuples → result;
else if ($n.rid$ is a prefix of $q.qk$ or $q.qk$ is a prefix of $n.r - id$ and q.type = "single")
start query processing aggProc(q);
search in $n$'s storage for tuples matching the queried value, retrieve all matching tuples → result
return (result)

**nexthop(*n*, *q*)**

For each neighbor $n_i$ of $n$,
// subtree of q.qk generally means the qk is a prefix of the sensor's vid or rid.
if (($n_i$.r-id or v-id belongs to subtree of q.qk) or ($n_i$.parent or $n_i$.child belongs to subtree of q.qk)) AND ($length(n_i.id) \leq length(n.id)$)
forward q to $n_i$ and return i;
else
$\delta(n_i, q) = sqrt((n_i.r - id_x - q.qk_x)^2 + (n_i.r - id_y - q.qk_y)^2)$
*next*=$\{\{n_i\} \mid \delta(n_i, q) = min(\delta(n_j, q), \delta(n, q))\}$
if *next* != *n*, return(*next*)
else
{*isCloser* = false; i = 1
while (!*isCloser*)
{send message *iterativeBFS*(q,n,i) to all neighbors $n_i$ of *n*
if (one *isCloserBFS*($q, n_i, i$) messages received from $n_i$ has the value "negative") *isCloser* = true; i++ }
} return($n_i$)

**Figure 18 Queryrouting and nexthop algorithms.**

## 5. Query Result Aggregation

### i. Aggregate Operators

Once the query gets to the query root, the second phase of query processing, i.e., query result aggregation phase, begins. Before we go into discussion, we will explore the different types of aggregate operators and how our AggIndex structure is utilized to process each of them.

[18] lists the following common attributes when classifying aggregation in sensor networks: (1) Duplicate sensitivity, which specifies whether an aggregate function will return the same result when there are duplicate values. Examples are MEDIAN, AVERAGE, and COUNT. On the contrary, MIN, MAX, and COUNT DISTINCT are insensitive. (2) Exemplary/nonexemplary: Exemplary aggregations (such as MIN, MAX, and MEDIAN) always return a representative value while nonexemplary aggregates (such as AVERAGE and COUNT) perform some calculation over the entire data space and return the calculated value. (3) Monotonic aggregates allow early testing of predicates in the network, for example, MAX and MIN. Significant savings in the overall number of messages sent through the network is possible by knowing in advance the location of a node or a set of nodes that hold such values. (4) Partial state requirements. Aggregates such as SUM and COUNT require partial state records that are the same size as the final aggregate. The AVERAGE function requires partial state records containing two values, i.e., both the SUM and the COUNT.

For aggregate operators such as MIN and MAX, the intrinsic characteristics of our indexing structure make the query processing rather straightforward. Because the r-id corresponds to the data region with a definite range of values, the result is stored at the sensor with the smallest or largest r-id within the range of the query. Therefore, query processing can be reduced to: (1) geographic routing of the query to the sensor that has the smallest or largest r-id within the query range; (2) search at the sensor; and (3) result delivery from this sensor to the query sink node. It is not necessary to search the whole network. Significant saving of energy results from using the indexing structure by the early knowledge of result-holding sensors. The pseudo-code of the search algorithm is shown in Figure 19.

```
SearchValue(range, searchtype)
_____________________________________________
if (searchtype == ”MIN” or ”MAX”)
{frid = the smallest or largest r-id calculated from the range;
sendquery(searchtype,rid) to the sensor holding rid;
}
```

**Figure 19 AggIndex Tree: Min-Max Search Pseudo-code**

Next, the AggIndex tree is exploited for processing queries with non-exemplary aggregate operators such as SUM, AVERAGE and COUNT.

### ii. Query Result Aggregation Algorithms using the AggIndex

Upon receiving a query, a sensor performs a search for matching tuples. Results from each sensor are then assembled or aggregated to generate a final result. This may be done at a designated query sink node. The disadvantage is the cost. An alternative would be to assemble or aggregate in network at the result holding sensors, instead, before reaching the query sink node. This approach is not only energy efficient, but it also causes less congestion. We found the latter approach more suitable for AggIndex.

**Definition 12** *A partial result of a query q, denoted* $<P(q)> = <P(q)_1, \ldots, P(q)_e>$*, is defined as an incomplete collection of e state records, which will be used in calculating the final result of the query. Partial results are aggregated by applying a merging function f on p partial results:* $<P'(q)> = f(<P_1(q)>, \ldots, <P_p(q)>)$.

Aggregation of partial results in a network often utilizes a hierarchic tree. In many previous approaches such as TAG [31], a routing tree rooted at the query issuing node is constructed and used to perform in-network aggregation. Such trees are often constructed by incrementally broadcasting a query message across the network and assigning tree levels to each node during the broadcasting. Although their implementation is relatively easy, they do not achieve an efficient solution for most networks, and they especially do not provide a viable solution for mobile environments. For example, long running queries are used on the same tree to perform aggregation. However, with topology changes in a mobile environment,

every query routing tree needs constant reconstruction to maintain the validity of the query aggregation.

Our aggregation scheme utilizes the AggIndex structure for query aggregation. The algorithm for processing a query q using AggIndex tree is described below. Its corresponding pseudo-code is shown in Figure 20.

```
aggProc(q)
_____________________________________________
if (q.qk in self.vidlist) declare q.aggroot = self.rid; send q to self.children
                compute pr if haven't, await partial result message from children
                combine pr to partial result, send final result to q.destination
else if (q.qk not in self.vidlist or self.rid) and (q.aggroot == null)
     then calculate partial result pr, forward q to ancestors by using nexthop(n,q).
     else if (q.qk not in self.vidlist) and (q.aggroot != null)
          then compute pr if haven't, await partialresult message from children
                combine pr to partial result, send partial result to self.parent
if (self.rid is not prefix of q.qk) | (q.qk is not prefix of self.rid) do nothing
```

**Figure 20 Aggregate query result processing.**

A query issuer first generates a query q and sends it out into the network. Each query q has a query key qk calculated from the range of the query as previously described. Query q, along with its qk is routed toward its aggregate root, which a tree node uniquely mapped to qk in the AggIndex tree.

**Definition 13** *The binary representation of qk is either equal to or exactly one of the v-ids or r-ids at the n is its max prefix. We refer to this node as the aggregate root for query q. In essence, the aggregate root is the query root for aggregate queries and is*

*computed in the same way as in other range queries. Only the sub-tree from the aggregation root will contain relevant data queried by q.*

As seen in Figure 20's algorithm aggProc, when a node n discovers a match between one of its ids and qk upon receiving q, it declares itself aggregation root of q and merges its own information to q. The issuer then sends out q along its sub-trees in the AggIndex tree. Query q propagates down the sub-tree until leaves are reached. The nodes examine their own local storage for matches and perform partial result aggregation if necessary. Some nodes already searched and processed results while the query was being routed before the aggregate root was reached. That would have saved some time and energy for the query result aggregation phase and those nodes can immediately deliver back the results upon request. After the partial results are computed they are propagated up along the tree to the parent nodes. Nodes of each level in the sub-tree await the partial results from all their children, merge the partial results into its own partial results, and send it up to the next level. The aggregate root, upon receiving the complete result, will post-process the aggregation result and route a query answer back to the query issuer using the same mechanism for query routing discussed earlier. When grouping predicates are present, each partial result is tagged with a group ID, and only those with the same group ID may be aggregated at intermediate nodes.

**Example** Consider again in the network of the use case example where the two domains D1 and D2 are pulse rate (40 beats/minute – 160 beats/minute) and systolic blood pressure (70mm Hg – 230mm Hg). An exemplary aggregate query q can be "SELECT COUNT(*) FROM complex-readings WHERE diastolic < 70 and systolic >

150." During processing, the query range (40 – 70, 150 – 230) is normalized to (0.25 – 0.5, 0.5 – 1), which is represented in binary form by (0.01, 0.1). The query key qk is computed to be 011. In Figure 13, this query is directed to nodes with r-id 0111 and 0110. Since node 0110 holds the v-id 011, it is declared the aggregate root for q. Query q is then propagated down its subtree, in this case just node 0111. The aggregate root 0110, upon receiving from node 0111 its partial result 6, combines it with its own partial result 4 in its local memory, and then delivers the complete count result 10 to the query issuer enclosed in q.

## E. Analysis

Let N refers to the network size and qp refers to the ratio of the query range over the entire data range. The cost to resolve a query consists of: **(**1**)** Routing cost for directing a query to the vicinity of the query destination and **(**2**)** the cost for retrieving matching tuples from all destinations. For **(**1**)**, the average cost in our case is $\sqrt{N}$ hops for dense networks. For **(**2**)**, the cost depends on the number of destination nodes, which is approximately N * qp, plus local searching cost.

**Definition 14** *A logical edge is an "edge" in the r-id tree or the AggIndex tree which connects a parent node with id I to its child with id Ib, where b is a bit **(**0 or 1**)**. A physical edge is an edge between sensors if they are within each other's direct communication range. A logical edge can be 0 hop **(**in which case a node occupies ids of both parent and child**)**, 1 hop **(**when there is a physical edge between the two nodes**)**, or multiple hops **(**when there is no direct physical edge between the two nodes**)**.*

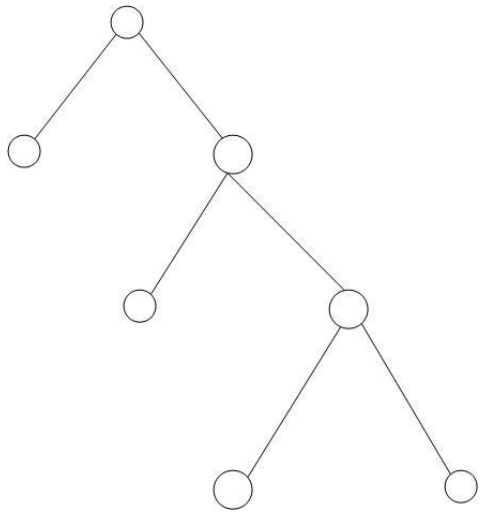
When a logical edge has multiple hops, we want to minimize the traversal cost such as number of hops and/or distance by following the shortest distance (ideally, a straight line from location A to location B) as much as possible.

**Discussion 2** In the case of a degenerated AggIndex logical tree (worst case scenario), whose depth is O(N), see Figure 21(a) for an example. The corresponding physical space partition, see Figure 21(b), is skewed but search cost needs not be O(N). As the graph shows, the upper-right corner of Figure 21(b) is much denser, contains more space partitions, and nodes tend to have higher degrees. Although the logical tree has a height of O(N) in Figure 21(a), in the physical world, the same leaf can be reached from the root in significantly less than O(N) hops in dense environments. Many hops can be skipped using greedy forwarding.
In the degenerated case where the logical tree is a degenerated tree of depth O**(**N**).**

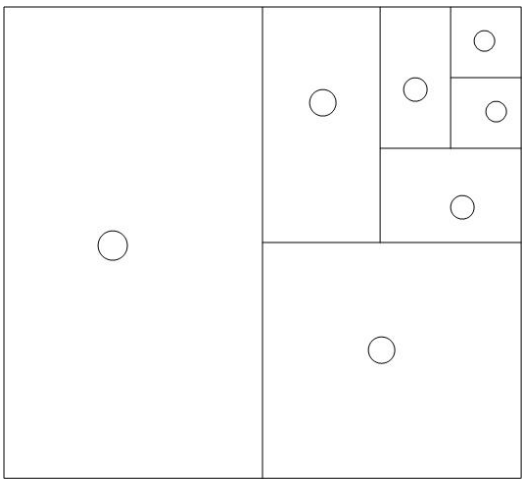
**Theorem 2** *Let L be the length of the physical held, and $r_h$ be the average hop Euclidian distance toward the destination. In the worse case, the path length between a root and the deepest leaf is 5L, and has 5L/$r_h$ nodes along the way.*

**Proof** In the degenerated case of discussion 2, the path length between a root and the deepest leaf **(**of depth O**(**N**))** in the Aggindex tree is at most $\sqrt{2}$L + $\sqrt{5}$/2L +$\sqrt{2}$/2L + **…** + $\frac{\sqrt{2}}{2^{n/2}}$ < **(**$\sqrt{2}$+ $\sqrt{5}$/2**)**L * $\frac{1}{1-1/2}$ = **(**2$\sqrt{2}$ + $\sqrt{5}$**)**L ≈ 5L. As the illustrating example in Figure 22, the worst case scenario creates a "zigzag" path from root to leaf, where each logical edge has the longest possible distance.

Similar to discussion 2, in the worst case of a balanced tree, a path from root to leaf in AggIndex is less than 5L in distance, and has less than $5L/r_h$ nodes along the way. □



**(a) Degenerated case logical AggIndex tree example**



**(b) Degenerated case example's corresponding physical space**

**Figure 21 A degenerated AggIndex tree example.**

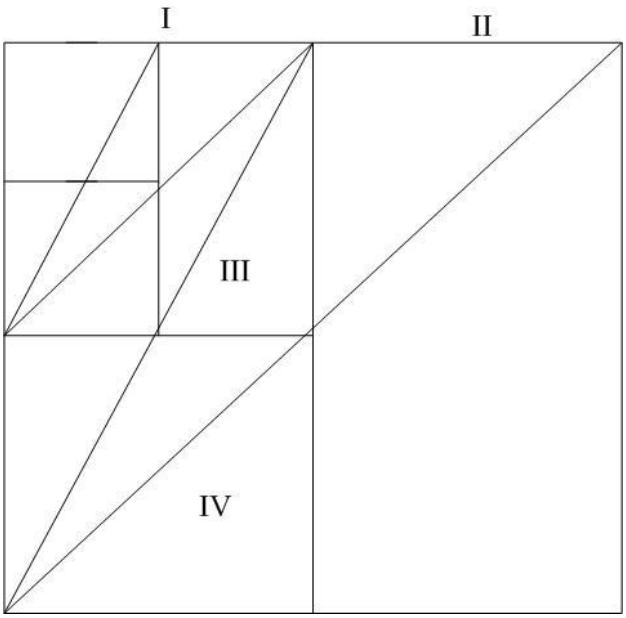



**Figure 22 Zigzag formation of a worst case AGGINDEX tree.**

**Theorem 3** *On average, the path length from root to leaf in AggIndex ≤ 2.266L, and is bounded by 2.266L/$r_h$ hops along the way.*

**Proof** On average, a logical tree is balanced and has height O**(**logn**)**. Therefore, in a network with n nodes, average length of r-id is logn. The path length is on average $\left(\frac{1+\sqrt{2}}{4}+\frac{2+\sqrt{5}}{8}\right)\mathrm{L}*+\frac{1}{2}\left(\frac{1+\sqrt{2}}{4}+\frac{2+\sqrt{5}}{8}\right)\mathrm{L}+\cdots+\frac{1}{2^{0.5\log n}}*\left(\frac{1+\sqrt{2}}{4}+\frac{2+\sqrt{5}}{8}\right)\mathrm{L}$**,** where $\left(\frac{1+\sqrt{2}}{4}\right)\mathrm{L}$ is the average distance between nodes in I and II, and $\left(\frac{2+\sqrt{5}}{8}\right)\mathrm{L}$ is the average distance between nodes in III and IV.The above formula is equal to or less than $\left(\frac{1+\sqrt{2}}{4}+\frac{2+\sqrt{5}}{8}\right)\mathrm{L}*\frac{1}{1-1/2}=\left(1+\frac{\sqrt{2}}{2}+\frac{\sqrt{5}}{4}\right)\mathrm{L}\approx 2.266\mathrm{L}$. Also, the number of hops is roughly bounded by 2.266L/rh. □

**Discussion 3** Consider the underlying physical space only, two nodes randomly selected from the field have on average a distance O(L). In the AggIndex, a tree path between arbitrary two nodes is, as seen in the theorems just discussed, a small constant of O(L) even in the worst case, which means it only has a constant spread factor over the physical space. This is a good result in correspondence to Theorem 1's locality property of the AggIndex scheme. According to Theorem 1, the mapping between physical and data spaces preserve C-locality where C is a constant.

**Discussion 3** In order for the search cost to be equal to or less than logN hops, $2.266L/r_h \leq \log N$, $r_h$ has to be $\geq 2.266L/\log N$.

**Discussion 4** The message cost to resolve a query consists of routing cost for directing a query to the vicinity of the query destination and the cost for retrieving matching tuples from all destinations. The routing cost can be divided into: (i) routing tree setup cost per query; and (ii) actual routing cost per query. For the class of queries such as TAG [31] that does not use indexing scheme, a routing tree spanning the network will be constructed for each general query. Therefore, the message cost is 2N per query. The message cost for indexing scheme without any preset aggregation tree, is $\frac{P1}{rh} + O((N * qp)^2) + N * qp * \frac{P1}{rh}$. Here the number of result holding nodes is approximately N*qp, where N refers to the network size and qp refers to the ratio of the query range over the entire data range, and Pl is the average path length in the network. This is because message cost contains three parts: **(**1**)** the cost to forward to a query root is on average $Pl/r_h$; **(**2**)** the cost to split the query and distribute each split query to its destination within N*qp result holding sensors needs $O((N * qp)^2)$ messages; and **(**3**)** the cost to deliver each subquery back

to the query source using greedy forwarding will incur N * qp * P1 / $r_h$ messages. The message cost for AggIndex is even lower, because the tree setup cost is amortized by all queries in the network. □

**Theorem 4** *The average distance of a routing path using a combination of AggIndex and greedy forwarding is within the range of (1/3L; 2.266L). The average number of hops is thus bounded by (L/3$r_h$, 2.266L/$r_h$). The probability of encountering a query subtree on the routing path is on average qp for each hop and goes up as the query is further routed.*

**Lemma 4.1** The average distance between two random nodes in the network is 1/3L. This is because $E|X-Y| = \int_0^L \int_0^L |x-y| dxdy = 1/3L$.

**Lemma 4.2** The average distance between two random nodes in AggIndex tree is upperbounded by 2.266L, according to theorem 3.

**Lemma 4.3** The chance of encountering a node in the AggIndex subtree of a query increases with each step of the routing, thus lowering the overall probability of falling into a local maxima during routing. On average, the probability of meeting a node within the query subtree at hop j is qp+(1-qp)*p(j), p(j) = $\sum_{i=1}^{qp*N} P((\mathrm{d_i} - \sum_{k=1}^{j} r_h(i,k))/r_c) < 1)$. The probability can be approximated by qp+(1-qp)* p'(j), p'(j) = $\sum_{i=1}^{qp*N} P((\mathrm{d_i} - j * r_h/r_c) < 1)$. The probability of falling into a local maxima falls because every time a local maxima results, the AggIndex is checked for recovery, providing an alternative route to the destination.

**Proof of Lemma 4.3** Assuming a query is generated randomly, the probability for the query to be generated within the subtree of the result holding nodes is lowerbounded by qp. As defined earlier, qp is the ratio of the query range over the entire data range. In this situation, only AggIndex tree is needed for query routing and processing. For example, qp = 0.1. This means with at least 0.1 probability query processing can immediately begin once it is generated.

For a probability upperbounded by 1 − qp, the query is generated outside the subtree of the result holding nodes. Let $d_i$ be the initial distance between the source node and each of the result holding nodes in the subtree. The probability of meeting a node within the query subtree at hop j p(j) = $\sum_{i=1}^{qp*N} P((d_i - \sum_{k=1}^{j} r_h(i,k))/r_c) < 1)$. $r_h(i,k)$ is the shortened distance to node i at hop k. To simply this, use the average hop distance $r_h$ to approximate each $r_h(i,k)$. The simplified approximated probability thus becomes p'(j) = $\sum_{i=1}^{qp*N} P((d_i - j * r_h/r_c) < 1)$. Obviously, the more hops the query takes, i.e., the closer it gets to the result holding nodes, the probability will rise and converges to 1. □

**Theorem 5** *In the degenerated (worst) case, the longest distance of a routing path using a combination of AggIndex and greedy forwarding is within the range of ($\sqrt{2}L$, 5L). The average hops is thus bounded by ($\sqrt{2}L/r_h$, $5L/r_h$).*

**Lemma 5.1** The longest distance between two random nodes in the network is $\sqrt{2}$L.

**Lemma 5.2** The longest distance between two random nodes in AggIndex tree is upperbounded by 5L. In the degenerated case according to theorem 2, each logical edge has the longest possible distance and forms a zigzag routing path with a total length upperbounded by 5L. □

**Discussion 5** The local maxima problem occurs when there is no node (by the current node's own knowledge) other than the node itself that is closer to the destination. Our conjecture is there is almost no practical case where greedy forwarding combined with using AggIndex in routing will fail to reach the query root or the subtree. Because the AggIndex is an overlay built in network and preserves some locality properties (Theorem 1 and Discussion 2), there is a significant lower chance of encountering the local maxima during the course of the routing. In addition, the added procedure of iBFS will help recovery from local maxima even in the worst case.

## F. Approximate Aggregations

In many scenarios, exact aggregate computations may not be necessary and are often too costly to be performed frequently. Approximate answers can reduce processing time and energy consumption significantly. Interestingly, we observe that the AggIndex tree is adapted to perform approximate queries. Since a r-id captures the data range mapped to it, it can be used to answer approximate queries by just estimating the average value stored at the sensor or the number of records stored at the sensor.

For example, for non-monotonic aggregate queries such as SUM, AggIndex tree can provide approximate result by exploring certain parts of the tree without the need to go to each individual sensor.

Let the number of data items stored at a sensor n be c(n). As discussed earlier, query q is directed to the aggregate root. The value of all tuples stored at each sensor can be approximated by the middle point of each sensor's data range indicated by r-id or through a histogram.

A partial aggregation result is approximated with the middle point value and c(n), and then propagated back along the tree. For example, with SUM, the partial result at n is obtained by multiplying the middle point value and c(n) and adding it to the partial result obtained from n's sub-tree. Clearly, approximate aggregation can respond very quickly to queries. It is especially beneficial if time and energy requirements are very stringent and accuracy can be sacrificed a little bit. Experiments show high accuracy can be achieved by approximate queries in our infrastructure.

# V. STARLET SIMULATION SYSTEM AND EXPERIMENTS

We developed an integrated simulation platform to study and compare algorithm performances in the areas of localization, event handling, network topology control, routing, and query processing in various types of sensor networks. Our research strategy and results are outlined in this chapter.

## A. Introduction of the Starlet System

In recent years a lot of research projects and a number of algorithms have been proposed for a wide variety of sensor network applications. Our research interests mainly cover areas in localization, indexing, routing, and query processing. A pressing need is to study the performances of various programs by simulation, which involves a lot of programming and debugging work. It is desirable to work with a simulation platform that is easy to use, portable, and free of bugs or loop holes. When a simulation platform contains bugs or loop holes, it becomes extremely difficult for users who are not developers of the platform to locate the problems, and even more difficult to fix the problems.

We have been actively searching for JAVA-based simulation platforms for years. We have tested a number of simulation platforms developed by student groups across the world. We classify all the platforms we have tried into two major classes:

(1) The first class of simulation platforms is implemented by a combination of

programming languages (such as JAVA or C++) and Tcl scripts. Examples include ns-2 [63] (implemented by C++ and OTcl) and J-Sim [64] (implemented by JAVA and Jacl). The simulation system itself is typically implemented as components in C++ or JAVA, while a simulation environment is usually created by gluing together components, setting initial conditions and parameters in Tcl scripts.

(2) The second class of simulation platforms is implemented by a single programming language such as JAVA or C++. Examples include Jist [65].

However, our tests with the existing simulation platforms are not entirely satisfactory. For most simulation platforms such as J-sim and Jist, the patches are slow to come out. It has been very difficult to implement our integrated solution for localization-based indexing on any of the previously mentioned platforms, as the coding involves a lot of cross-component integration and the stratification of those simulators proves to be cumbersome for the implementation of integrated solutions. Since we are mainly dealing with the application layer and a simplified routing layer in our algorithms, we want the underling layers to be as thin as possible so as to avoid unnecessary programming tasks. In addition, the Tcl scripts are often quite complicated to implement, requiring profound knowledge of the whole simulation package across all layers in order to set up the network environment properly. We also want to be able to test with large scale networks (as many as 40,000 nodes), which is well beyond the means of current completely layered simulation platforms such as ns-2. Therefore, the idea of building our own simulation platform is motivated by our unique research requirement, which is to build a system that has built-in solutions for various sensor networks (with either distributed or centralized control), and is

convenient for algorithm implementation and comparison. The simulation platform has to be as light-weighted as possible for scalability, and as full-fledged as possible to simulate a number of sensor specific operations.

Another motivation for building a simulation platform of our own is to provide a simple user interface so that people without any programming background will be able to use and test our algorithms. Our interface allows a user to create a network, create events, ask queries, and observe results dynamically through a user-friendly GUI. This will also help investors to quickly get the idea of the technology and observe the performances.

## B. System Design Methodology

The design of the system was completed in 2007 and published in [66]. Under an object-oriented design, we built sensors, a router, a gateway, queries, and events into objects. The top hierarchy is shown in FIG. 21, including a GUI and three classes of user-specified sensor networks.

### 1. Sensor Networks

Three major classes of sensor network organization methods are implemented:

(1) Distributed sensor network, where each sensor manages its own storage, routes packets, and answers query distributed;

(2) Central 1 network, where each sensor sends recorded events to a base station and does not execute local control of any data; and

(3) Central 2 (or semi-central), where each sensor manages its own storage, but has to send the data to the base station for query processing request.

Upon obtaining a number of parameters such as field size, sensor communication range, sensor-net type and number of sensors, a new network is initiated. Network topology is maintained by regular “hello” messages sent at every specified interval to advertise a node's existence among its neighborhood. Our simulation platform allows various sensor distributions. It can also take network configurations and queries from user input for online observation of the simulation process. Figure 28 shows the topology of a 600*600 (square meters), 100-node network.

Both static and mobile sensors are implemented in our platform. For mobile sensors, movement is modeled in a discrete way: A random number is generated in each iteration to determine the motion direction. A sensor has equal probability to move toward any of the four directions. This simulates a random walk in the 2-D space.

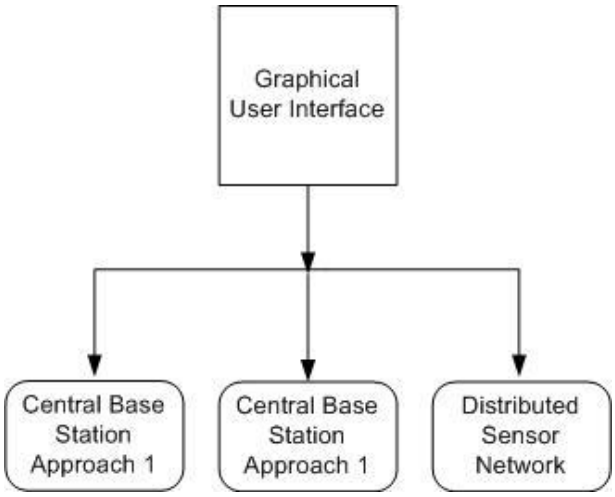



**Figure 21 Top hierarchy.**

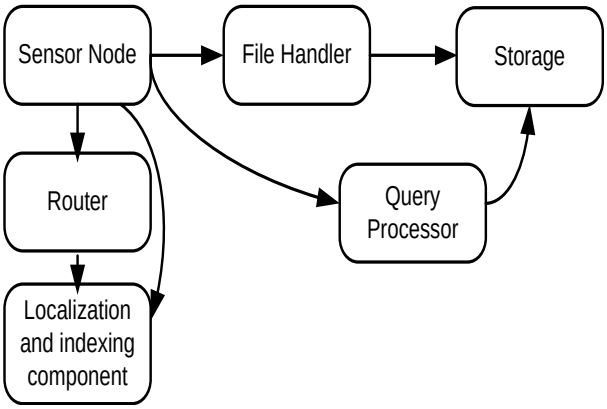



**Figure 22 Distributed approach.**

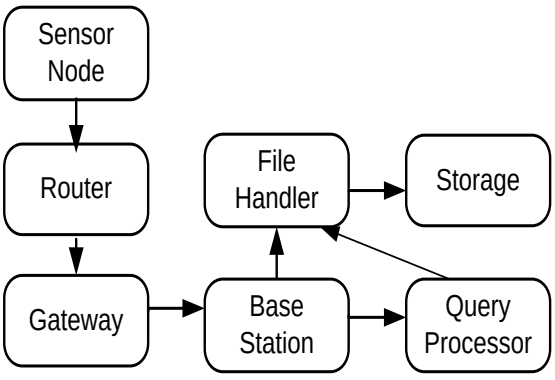



**Figure 23 Central base station approach 1.**

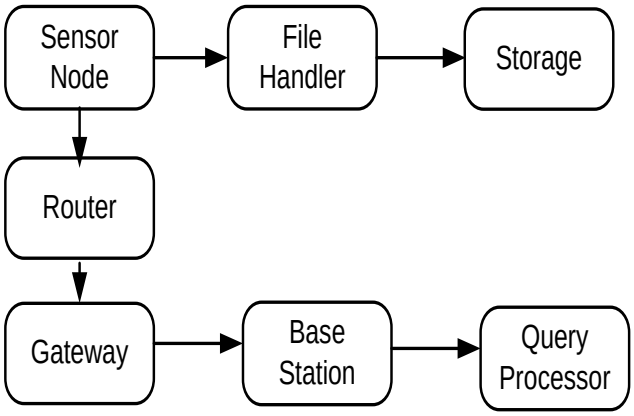



**Figure 24 Central base station approach 2.**

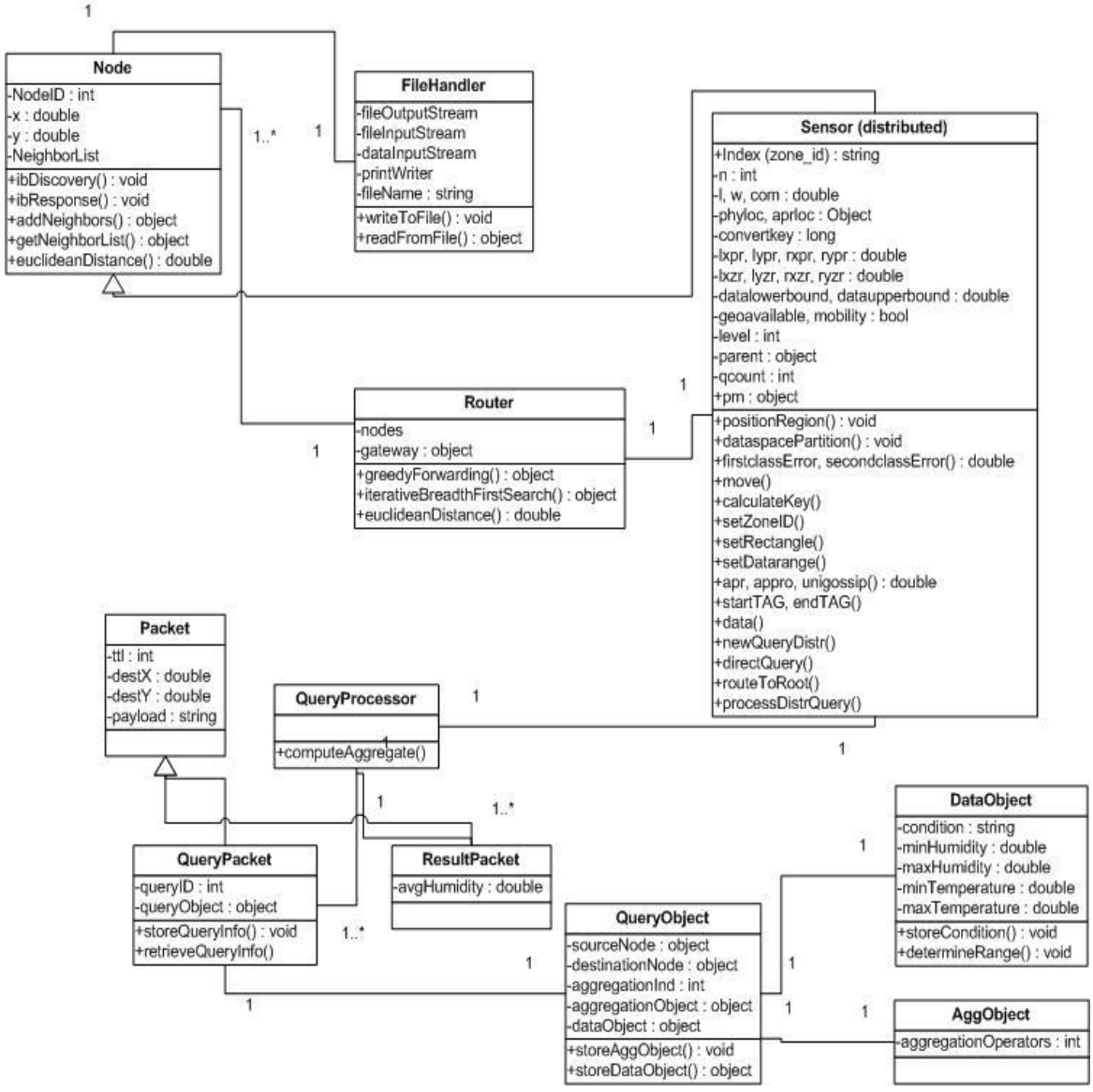



**Figure 25 Distributed approach class diagram.**

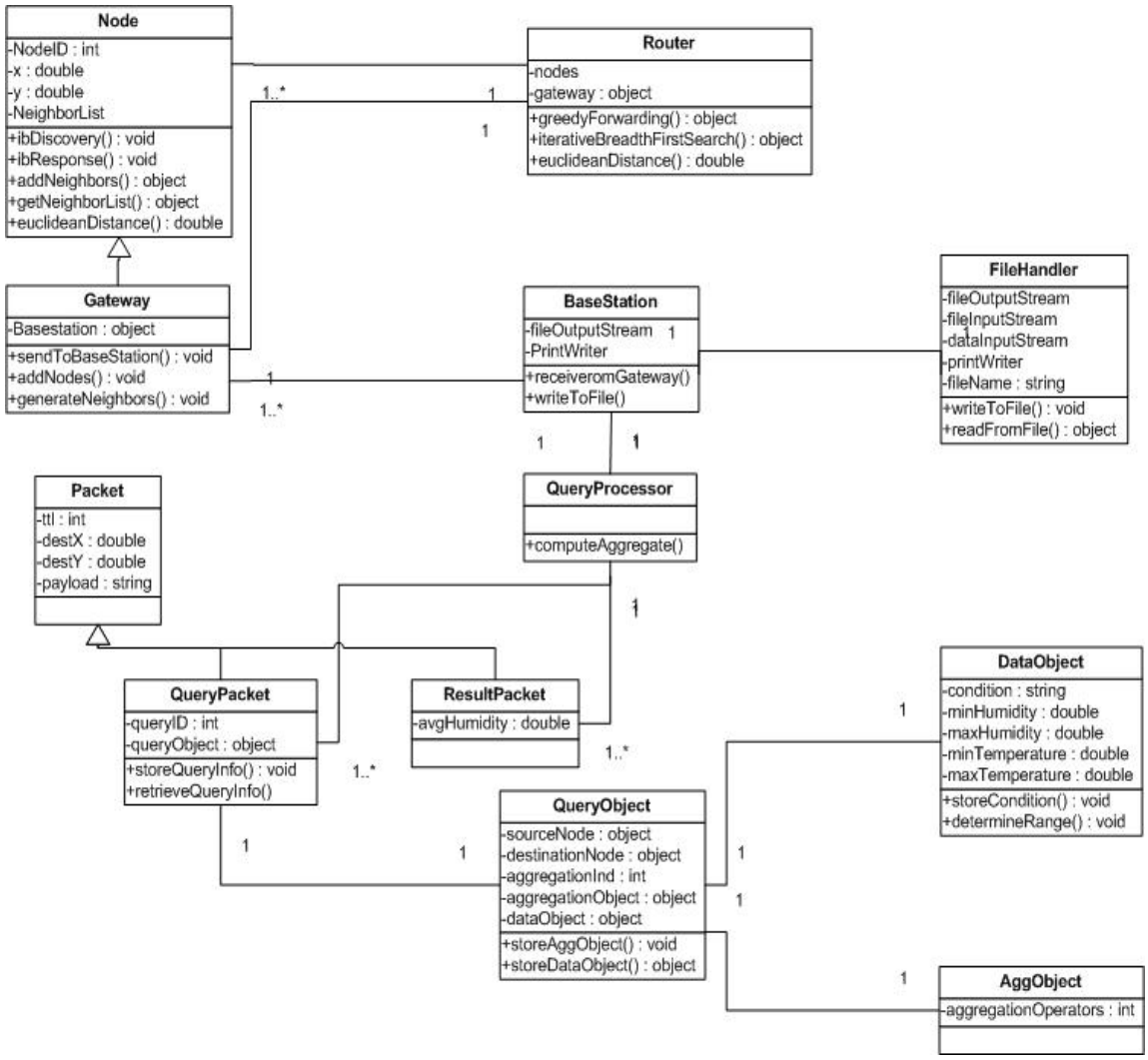



**Figure 26 Central 1 approach class diagram.**

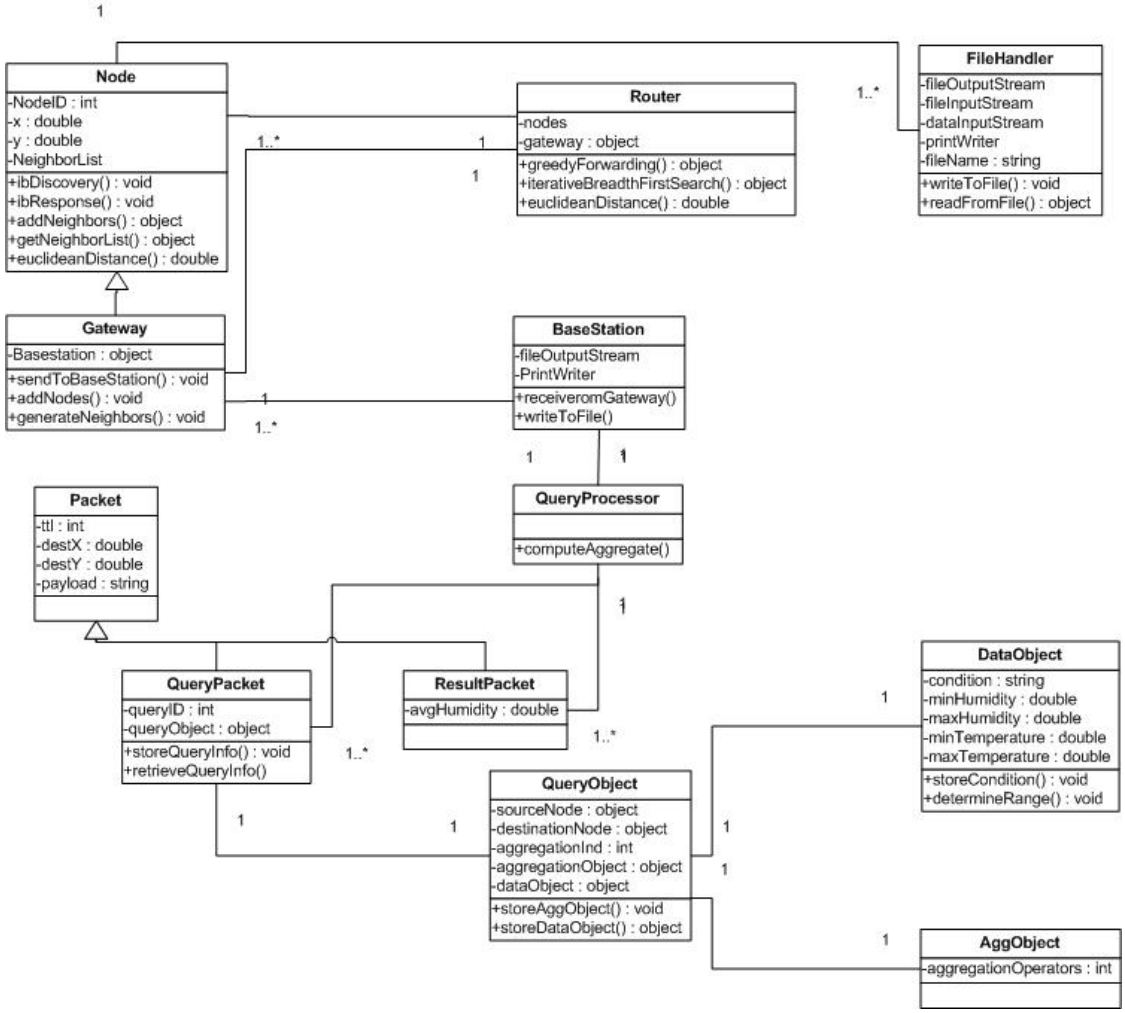



**Figure 27 Central 2 approach class diagram.**

## 2. Sensors

For simplicity, our simulator does not consider certain low-level properties of the network such as radio contention. At the lowest level, the architecture consists of sensor nodes. The sensor nodes perform general purpose computing and networking in addition to application-specific sensing tasks. A distributed sensor is a subclass of a general purpose node. The distributed sensor subclass implements functions such as location discovery, indexing, and in-network query processing. The relationship of the

sensor class with other system components is illustrated in Figure 22 - 24. Figure 22 shows the distributed sensor class directly interacting with all other components of the system, including event handling, storage, localization, query processing and routing.

Both centralized approaches consist of a gateway responsible for transmitting sensed data from nodes to the base-station. The base-station either stores all the data at its location (central 1) or gets the required data from the sensors once a query is issued (central 2). The base-station also has the ability to process the query and perform aggregation based on the query. Figure 23 shows the central 1 approach, where sensors have the least control – serving only as routers – while the base station deals with most of the operations above. Figure 24 shows the central 2 approach, where sensors have more control than the central 1 approach by having storage and event handling capabilities, but still rely on base station to process queries.

Figures 25 - 27 are the class diagrams for the distributed, central 1 and central 2 approaches respectively.

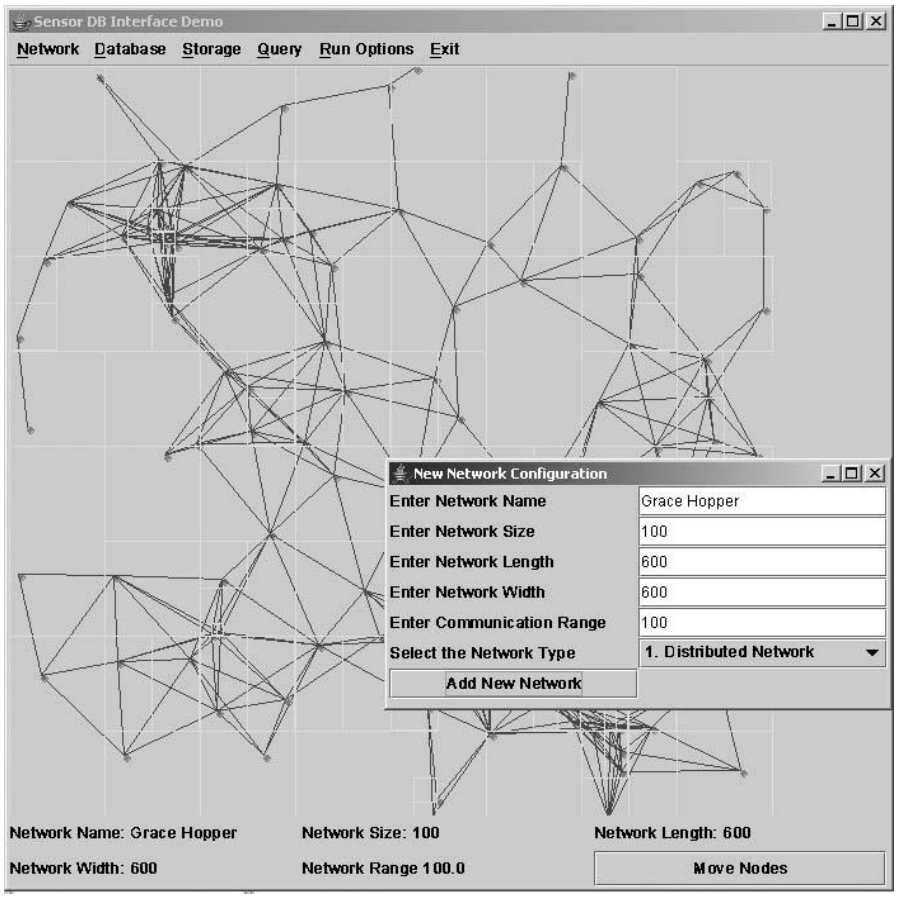



**Figure 28 Simulator interface – "NewNetwork" operation.**

### 3. Router

The router component provides packet routing and transmission throughout the network by greedy geographic forwarding and iterative breadth first search functions.

Several localization methods (described in Chapter II, III) are built into the distributed sensors in our simulator. The purpose of localization is to allow sensors without geographic capabilities to figure out their locations from other sensor's location information.

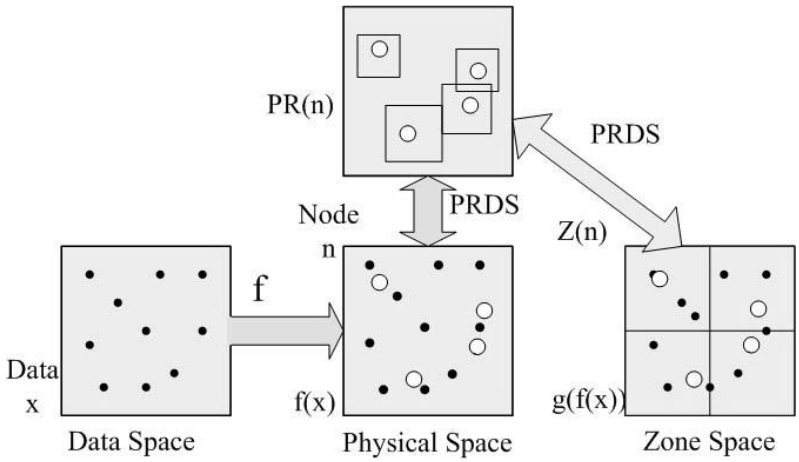



**Figure 29 PRDS localization at sensors**

Indexing is a data organization technique for managing distributed sensor storages. Indexing is implemented at the sensors. The simulator implements a localization-integrated indexing scheme that allows a network of sensors to initiate or re-organize among themselves. As a result, each sensor gets its share of the space. Figure 29 illustrates the partitioning process.

To illustrate the indexing scheme and data space partitioning, we built a GUI interface that can take inputs from users for sensor system configuration and show the organization result of the indexing by having each sensor showing its partition of the space. As seen in Figure 29, the exemplary network is shown with its topology and partitioned space, where each sensor manages a partition delineated by the rectangle enclosing it.

## 4. Query processor

Our project is targeted at providing a database query application paradigm in the wireless ad hoc sensor network. Some state-of-the-art query processing systems like TinyDB [35], which works on top of the TinyOS operating system, suffers from an external storage problem. It requires an external database connected to sensors at all times. In many real world cases, this is either not possible because of the bandwidth limitation (for example, in a giant panda forest reserve, communication to outside the network is extremely costly), or is a potential burden on sensors because the power consumption for query will be high (caused by flooding the network). By implementing our indexing structure in the database structure, we provide a storage solution to solve the above problems. Our query processing platform is built on the indexing structure to efficiently process queries.

An important feature of the simulation, query processor is implemented at each distributed sensor and also in the base station for centralized approaches. A query can be generated at any sensor node or the base station. Each query issued specifies a destination sensor node where the result needs to be sent. In the distributed approach, a query is directed to the range of sensors specified in the condition. In the centralized approaches, once the query is issued at a node, it is routed from that node to the base station via the gateway. In the central1 approach, event handling and storage is managed by the base station, hence data is retrieved from the storage at the base station and the query is processed on the data. In the central2 approach, since each node performs its own event handling and storage; the base station retrieves required

data from the sensor nodes, after which the query is processed at the base station based on the received data from the sensor nodes.

The processing of the query involves the following objects:-

1 DataObject – The DataObject is used to store the condition of the query. For example if the query is to find out the average humidity value given the range of temperature and humidity. The DataObject stores the range of humidity and temperature given by the user, which is utilized while processing the query.

2 QueryObject – The QueryObject is used to store the DataObject for the condition, the source node and the destination node IDs. It's also used to store the aggregation indicator and the aggregation object.

3 AggObject – The AggObject is used to store the aggregation operator that is requested by the user's query. For example, the aggregation operators that could be utilized are 1 for Sum, 2 for Average, 3 for Count, 4 for Max and 5 for Min.

4 QueryPacket – The QueryPacket is used to keep track of each query individually from its inception to the end. The QueryPacket consists of the QueryObject. For the distributed solution, the QueryPacket is routed from the source node to all of the nodes that potentially hold query results. For the centralized solutions, the QueryPacket is routed from the source node to the base station via the gateway. The basic reason for having a QueryPacket is to distinguish each query originating in the network uniquely, as communication in the network generally happens in the form of packets.

5 ResultPacket – The ResultPacket is used to store the result.

Each distributed sensor contains a query processing unit. The query-processing unit consists of all classes above. When a sensor node receives a query and discovers that it is within the range of that query, it will forward the query to neighboring sensors in the range and search its own storage for partial results. Once the sensor receives partial results from all neighbors it forwarded the query to and itself, it then merges the results and returns the merged result to the sensor that first notified the query. Once all results are merged, the sensor node in charge will send the result to the destination node using router's functionality.

For the centralized solutions, the base-station contains a query-processing unit. The query-processing unit consists of the query object and the query packet class. Once processed, the result is transmitted from the base-station to the gateway and from the gateway to the destination node using the router's functionality. The central base-station approach 2 is a semi-centralized approach for implementing the sensor network as a database. Each sensor manages its own data in a file. When a query is issued to a node, the node routes the query to the base-station via the gateway. The gateway is a subclass of the node class. The gateway class is responsible to transfer the packet to the base-station. The routing is done using the router's greedy forwarding and iterative breadth first search methodology to find the path along which the packet needs to be routed. Once the base-station receives the query, it tries to get the data from the nodes that store the data in the range of the query, using the gateway. The base-station makes use of the query-processing unit. The query-processing unit consists of the query object and the query packet class. Once processed, the result is transmitted from the base-station to the gateway and from the gateway to the destination node using the router's functionality.

### 5. File Handler and Event generator

The file handler component performs all file handling operations for sensors and base stations. Typical operations include reading data from a file, writing data to a file, opening and closing of files etc. In the distributed approach and central base station approach 2, data is managed by each sensor in storage or memory. In the central base station approach 1, data is written and read from storage at the base station.

The event generator component simulates data sensing events in the network. Each event can have a number of attributes, such as blood pressure and body temperature in a health care scenario. Users can also specify the number of events to be generated and the value range for each attribute. Three types of event generation methods are included in the simulator:

(1) Random event. Events taking random values are generated at randomly picked sensors. In the distributed approach and the central base station approach 2, the events are inserted as data into the storage of corresponding sensors, while in the center base station 1 approach, the events are sent through messages from the originating sensors to the base station for storage and further processing. A user can add user-specific distribution to the random event data generation to generate the types of events distribution he or she wants to experiment with.

(2) Even event, where data is generated at every sensor. In this case every sensor receives data. The data is generated in a linear pattern.

(3) Uniformly distributed event. In this case we generate data initially and store it in arrays. We then randomly chose data value from the arrays for the sensors. By doing so we ensure that we are distributing data across the whole data range.

## 6. Interface

The graphical user interface provides easy access for those users without profound knowledge of the system. Some exemplary operations include:

1. NewNetwork: A user will initiate this function in the GUI to setup the network by inputting parameters such as network type, field size, number of nodes, and communication ability.
2. DrawNode: Display locations of all sensors, network topology, and partitionss assigned to each sensor. The user can also view some selected routing paths.
3. Move: For a mobile network, this function will allow a portion of sensors to move in a specified way.
4. NewQuery: This function will allow a user to ask a query by inputting the query range, source and destination nodes. Results will be displayed through the interface.

As seen in Figure 28, a user can input network parameters to initiate a new network. The user can select from the three approaches for the new network. After the network is initiated, the user can perform functions such as generating events and asking queries from the interface.

## C. Experimental Results

When measuring performances, the period of time allotted for exchanging messages between two levels in a tree is called an epoch. The latency of a query result is dominated by the product of the epoch duration and the total number of epochs for a query to be answered since it was issued. By assuming the same epoch duration for different approaches, our comparison of latency is dominated by the number of epochs elapsed for a query to be answered.

We have a built-in performance metrics class to measure metrics such as the number of epochs, messages, packets, delivery ratio for a set of operations. The class is globally accessed and can be reset at all times. Some of the experimental results we obtained include:

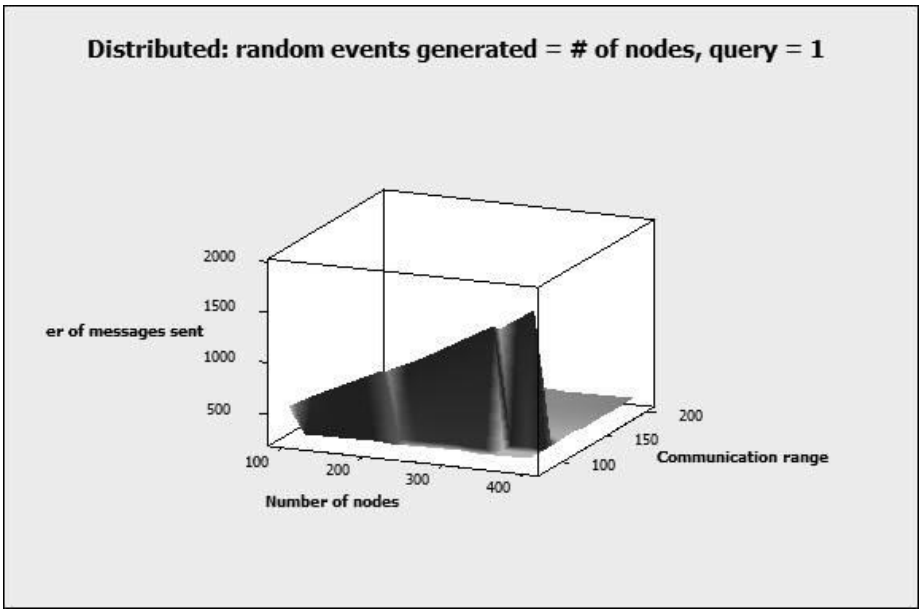



**Figure 30 Distributed approach - AggIndex (# of messages over N and $r_c$).**

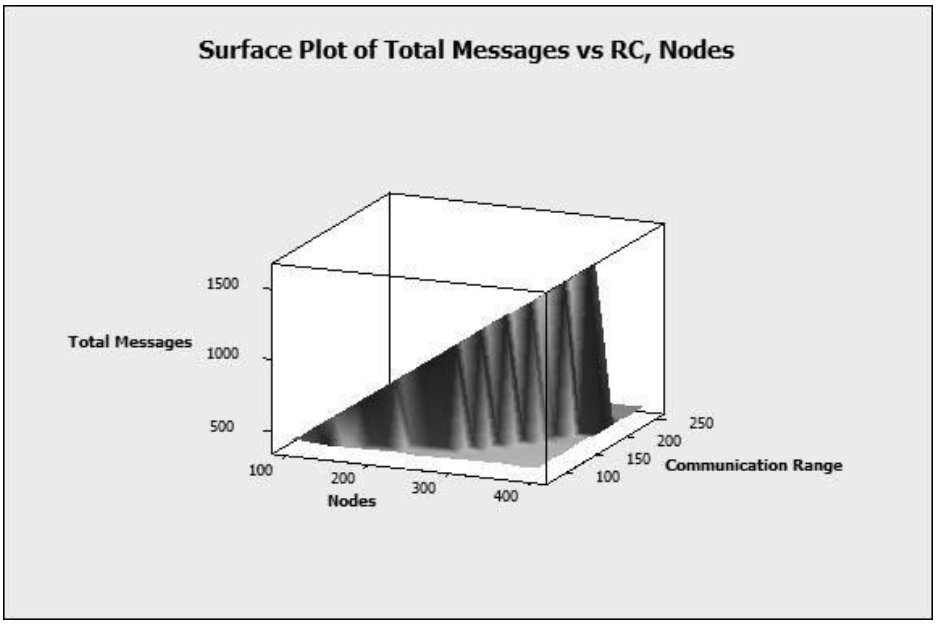



**Figure 31 Central 1 (# messages over N and $r_c$).**

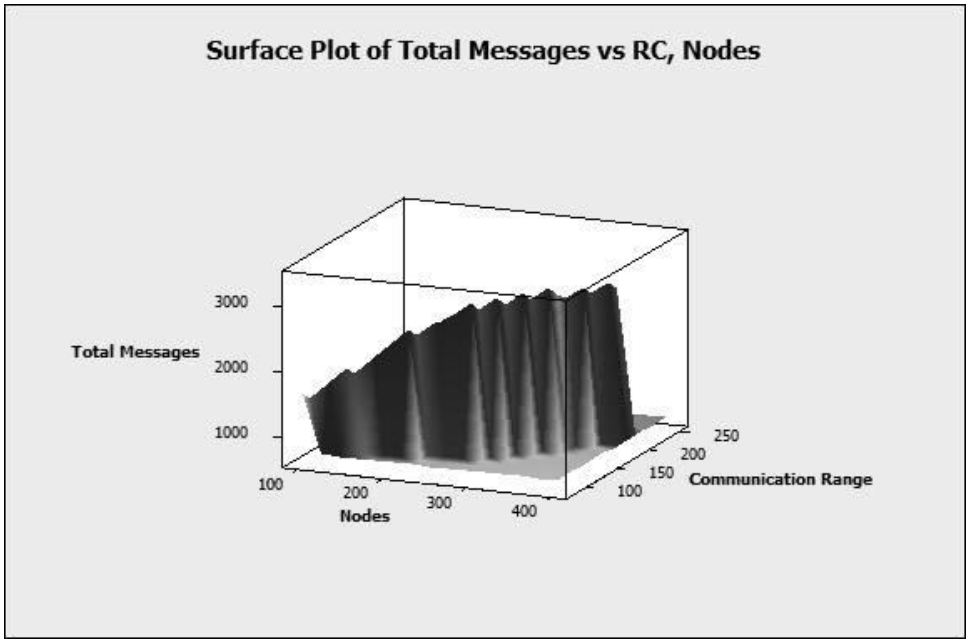



**Figure 32 Central 2 (semi-central) avg message over N and $r_c$.**

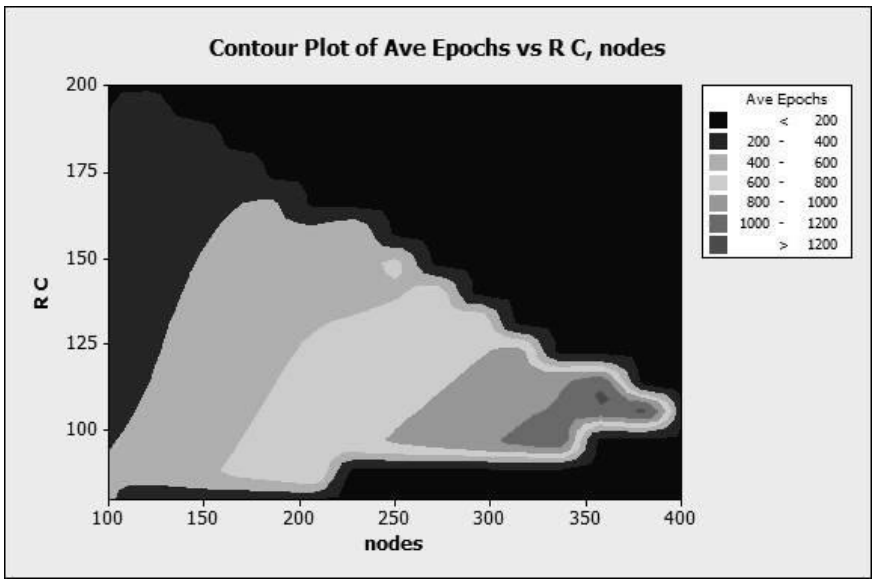



**Figure 33 Distributed approach - AggIndex (avg epochs over N and $r_c$).**

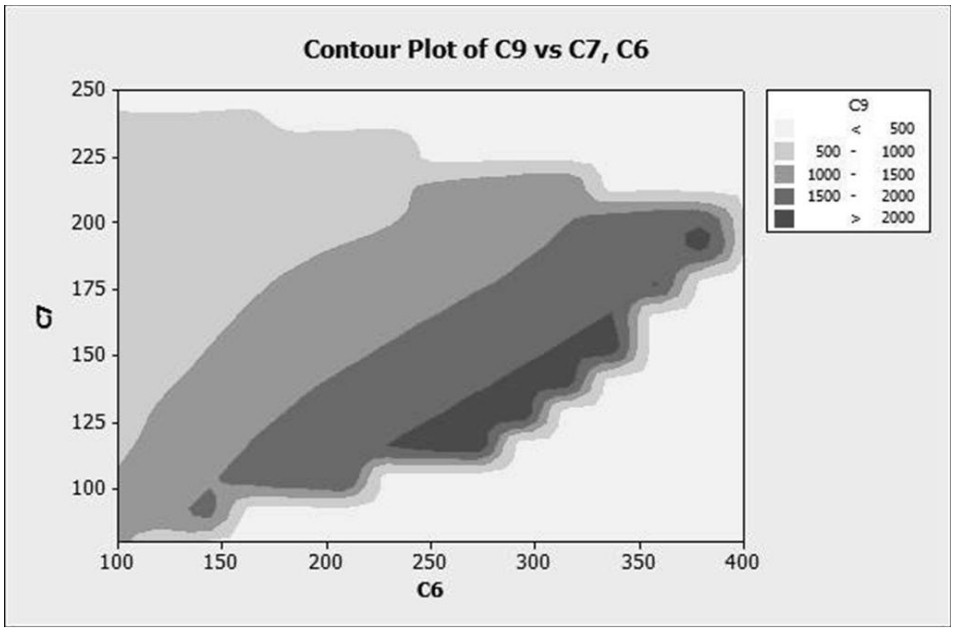



**Figure 34 Avg epochs over N and *rc* (semi-central).**

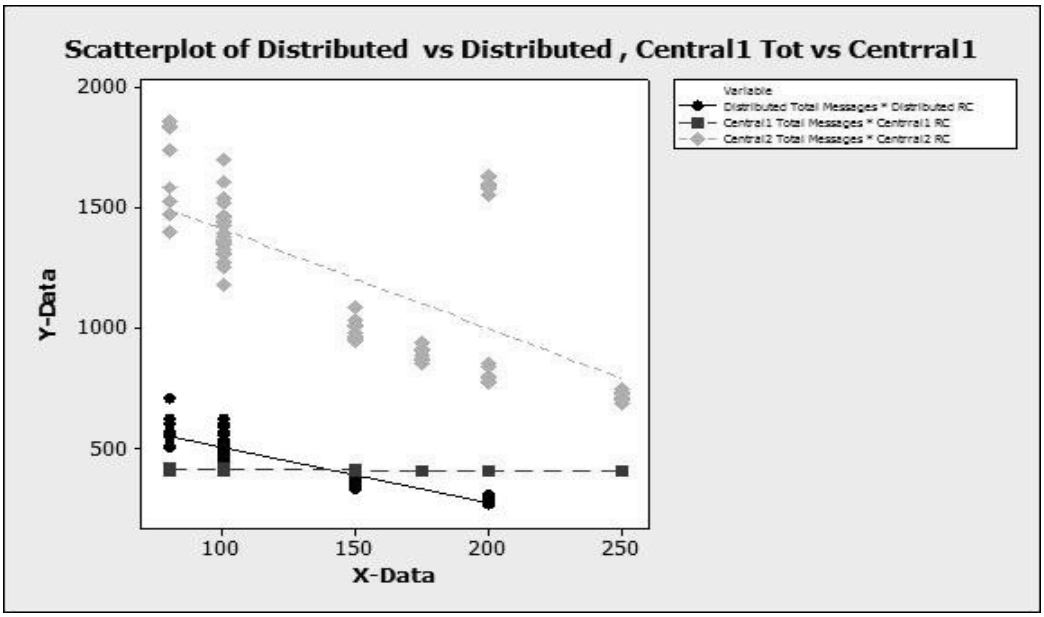



**Figure 35 Comparison of messages vs. communication range.**

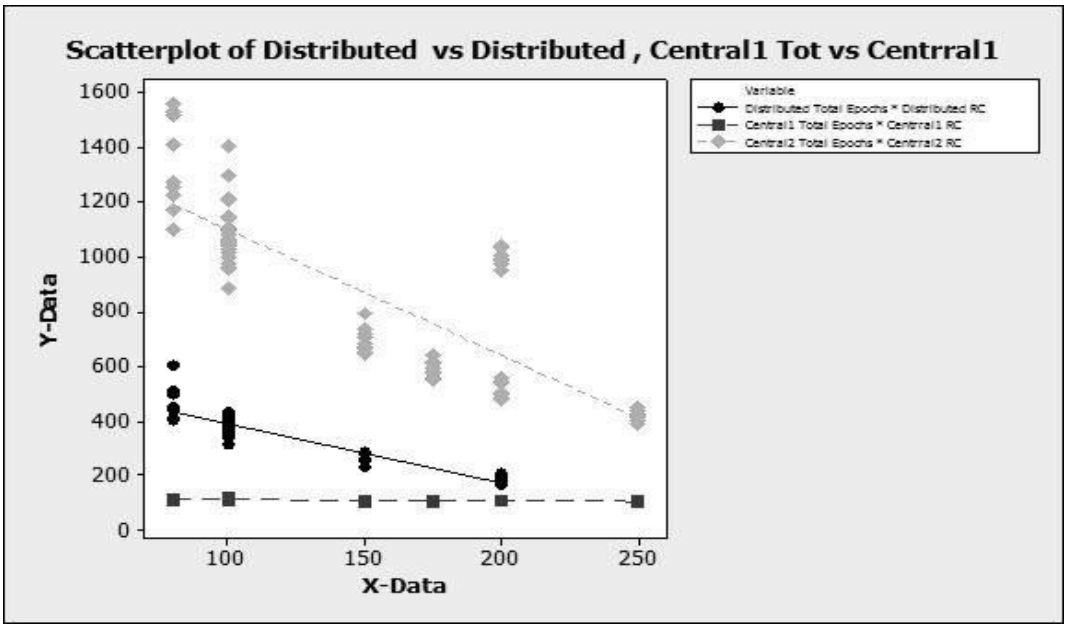



**Figure 36 Comparison of epochs vs. communication range.**

Figure 30 plots the contour graph for the average message cost for varying network sizes and communication ranges in the distributed approach. Each query set is simulated by 100 random data events and 1 aggregate query over a selected range. We have also conducted experiments on the same network settings and query sets in the central 1 and 2 approaches, as seen in Figure 31 and 32. The distributed approach incurs fewer messages (nearly 1/2) than the central 2 approach. This can be explained by that in-network query processing of the distributed approach incurs substantially fewer messages than the central 2 approach, where in the latter each sensor needs to be informed the query and shall send the data in its local storage to the base station for processing. The distributed approach incurs slightly higher number of messages than the central 1 approach in most cases. This can be explained by that the central 1 approach has the base station keep all the data and perform query processing, and thus saves the part of messages for in-network processing. However, the central 1 approach creates a bottleneck in the gateway near the base station. The network is thus pliable to congestion, causing the gateway node to deplete its power much faster than the other nodes.

Figure 33 plots the average number of epochs for the distributed approach on the same query sets. Figure 35 shows the comparison of message cost while the number of nodes remains constant. The graph shows that the central 1 approach is not affected by the changes in the communication range. The central 2 and the distributed approaches behave similarly as the number of messages decrease with the increase in rc. However, the number of messages is significantly higher for the central 2 approach than the distributed approach. Figure 36 shows the comparison of epochs while the number of nodes remains constant.

The AggIndex tree approach for query aggregation, which was proposed in Chapter IV, has been compared with gossip-based aggregation **(**GOSSIP**)** and the class of aggregation techniques on spanning-trees exemplified by TAG and Cougar. Query size is defined as the percentage of query-related records over all records. In the uniform case, query size is proportional to the number of sensors that answer the query.

The benefit of using the AggIndex approach over TAG and GOSSIP to answer monotonic aggregates such as MAX and MIN is obvious, because knowing in advance the nodes that hold such values significantly reduces query propagation cost. For non-monotonic aggregates such as SUM, we conducted experiments to compare different approaches with varying query size in latency and message cost spent per aggregate query, as shown Figures 37 and 38. Experimental results shown in these two figures were conducted with random networks of 100 nodes where SUM queries were issued by randomly selected nodes. TAG was simulated with the original scheme given in [31], which was surveyed in Chapter IV. For GOSSIP the uniform gossip method was simulated [37], which was also surveyed in Chapter IV.

Figure 37 shows the latency measured in the total number of epochs elapsed for answering a SUM query. For GOSSIP, we calculate the number of epochs elapsed until gossiping results fall within 95% of the correct result. Among these approaches, GOSSIP has the longest latency for all query sizes. This explains the slow convergence of gossiping protocols. TAG has an almost uniform latency for all query sizes. When a query covers most of the records in the network, TAG incurs shorter latency than our AggIndex approach. However, as query size drops, latency of

AggIndex drops significantly and comes close to TAG when query size is 5**%** of all records in the network. The latency of AggIndex eventually drops below TAG as query size further decreases. Also, by incorporating summary keeping and query caching, the latency for AggIndex could be further reduced.

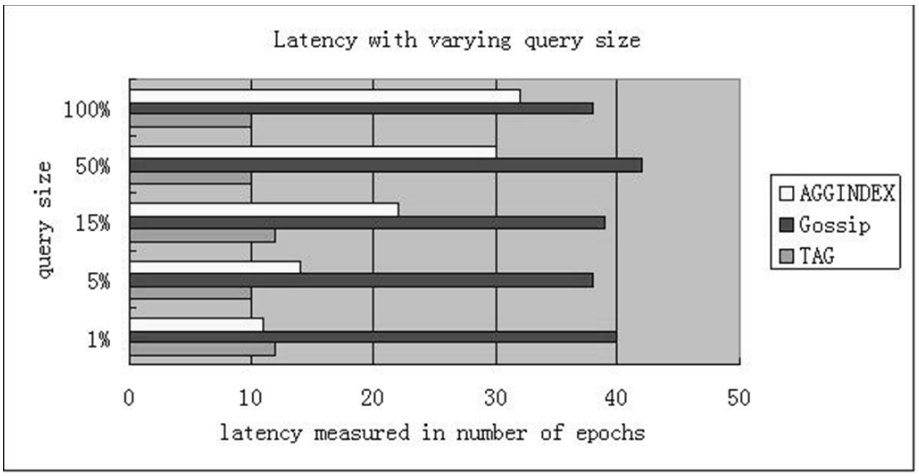



**Figure 37 Latency vs. varying network size**

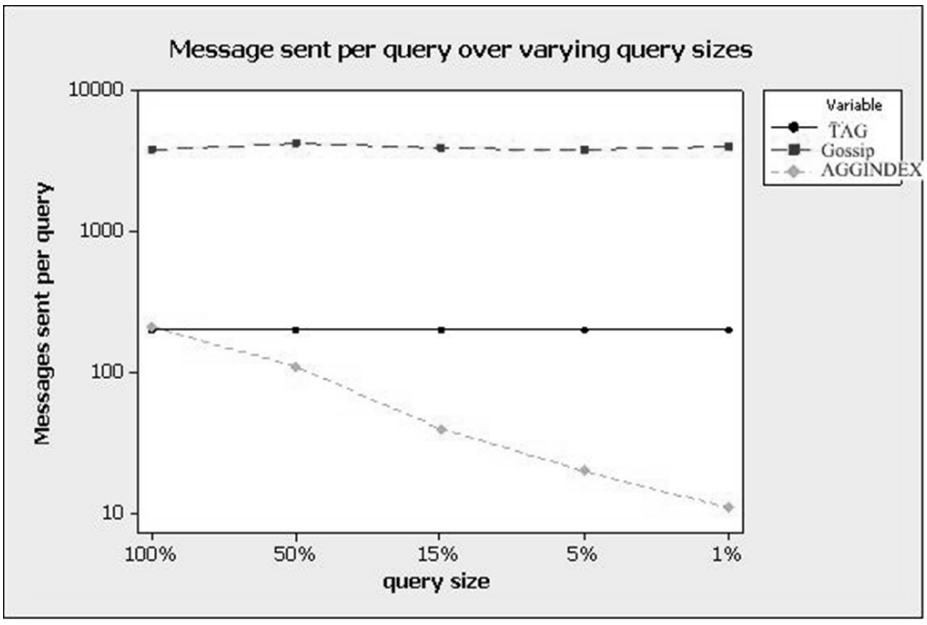



**Figure 38 Message cost vs. varying query size.**

Figure 38 shows total number of messages sent per SUM query. Again the number of messages sent by GOSSIP is calculated as the total number of messages sent until the gossiping result is within 95**%** of the correct result. GOSSIP incurs exponentially more messages **(**as shown in the figure on the logarithmic Y-axis**)**. This can be explained by the fact that every node in GOSSIP needs to send a message at the end of each epoch, while in TAG and AggIndex, for most of the epochs a node simply waits for the message from its next level without sending a message. Message cost for query sizes up to 100**%** of all records is similar for TAG and AggIndex. As the query size decreases, AggIndex tends to outperform TAG by requiring only $1/\sqrt{n}$ of the total messages of TAG **(**here the network size is 100**))**.

Go back to Figure 30, it plots the average message cost for varying network sizes and communication ranges in the AggIndex approach. Each query set is simulated by 100 random data events and 1 aggregate query over a selected range. We also conducted experiments on the same network setting and query sets in a "semi-centralized" query processing approach. The semi-centralized query processing approach keeps a local storage at each sensor, but has the sensor send relevant data to a base station whenever a query is issued. Those data are sent by packages through the network, and the base station receives the data through a gateway node, which is also a sensor in the network. Obviously the AggIndex approach incurs fewer messages (nearly 1/2) than the semi-centralized approach. This can be explained by that the in-network query processing of the AggIndex approach incurs substantially less messages than the semi-centralized approach, where each sensor need to be informed the query and shall send the data in the local storage in a reply message to the base station. Figure 33 plots the average epochs elapsed from the time a query is issued

until the time the result is delivered to the query sink in our approach. Again, the same set of query is used here. The comparison between our approach and the semi-distributed approach (Figure 34) shows that the AggIndex approach incurs significantly less epochs among the two.

## CITED REFERENCES


[1] X. Li, Y.J. Kim, R. Govindan, and W. Hong, "Multi-dimensional Range Queries in Sensor Networks," *Proceedings of the First ACM Conference on Embedded Network Sensor Systems (SenSys),* Los Angeles, California, USA: 2003, pp. 63-75.

[2] T. Moscibroda, R. O'Dell, M. Wattenhofer, and R. Wattenhofer, "Virtual Coordinates for Ad hoc Sensor Networks," Philadelphia, Pennsylvania, USA: 2004.

[3] C.D. Whitehouse, "The design of Calamari: an Ad-hoc Localization System for Sensor Networks," Master Thesis, University of California at Berkeley, 2002.

[4] A. Savvides, C. Han, and M.B. Strivastava, "Dynamic fine-grained localization in Ad-Hoc networks of sensors," *Proceedings of the 7th annual international conference on Mobile computing and networking,* New York, NY, USA: ACM, 2001, pp. 166–179.

[5] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.,* vol. 30, Jun. 1998, pp. 170–231.

[6] S.I. Roumeliotis and G.A. Bekey, "Collective Localization: A Distributed Kalman Filter Approach to Localization of Groups of Mobile Robots," *ICRA*, 2000, pp. 2958-2965.

[7] S. Capkun, M. Hamdi, and J. Hubaux, "GPS-free positioning in Mobile Ad-Hoc networks," *Proceedings of the 34th Annual Hawaii International Conference on System Sciences ( HICSS-34),* 2001, pp. 3481-3490.

[8] A. Galstyan, B. Krishnamachari, K. Lerman, and S. Pattem, "Distributed online localization in sensor networks using a moving target," *Proceedings of the 3rd international symposium on Information processing in sensor networks,* New York, NY, USA: ACM, 2004, pp. 61–70.

[9] C. Taylor, A. Rahimi, J. Bachrach, and H. Shrobe, *Simultaneous localization and tracking in an ad hoc sensor network*, MIT, 2005.

[10] A. Savvides, W.L. Garber, O.L. Moses, and M.B. Srivastava, "An analysis of error inducing parameters in multihop sensor node localization," *IEEE Transactions on Mobile Computing,* vol. 4, 2005, pp. 567-577.

[11] Krishna Kant Chintalapudi, A. Dhariwal, R. Govindan, and G. Sukhatme, *On the Feasibility of Ad-Hoc Localization Systems,*, Computer Science Department, University of South California, 2003.

[12] C. Savarese, J.M. Rabaey, and K. Langendoen, “Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks,” *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA: USENIX Association, 2002, pp. 317–327.

[13] K. Whitehouse and D. Culler, “Calibration as parameter estimation in sensor networks,” *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, New York, NY, USA: ACM, 2002, pp. 59–67.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA: ACM, 2001, pp. 161–172.

[15] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA: ACM, 2001, pp. 149–160.

[16] A.M. Ouksel and G. Moro, “G-Grid: A Class of Scalable and Self-organizing Data Structures for Multi-dimensional quering and content routing in P2P network,” *Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC'2003)*, 2003, pp. 123-137.

[17] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker, “GHT: a geographic hash table for data-centric storage,” *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, New York, NY, USA: ACM, 2002, pp. 78–87.

[18] L. Xiao and A.M. Ouksel, “Tolerance of localization imprecision in efficiently managing mobile sensor databases,” *ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, 2005, pp. 25-32.

[19]A. Ouksel and L. Xiao, “Method and system for managing a network of sensors,” U.S. Patent 7,840,353, November 23, 2010.

[20] Z. Zhou, Z. Peng, J. Cui, Z. Shi, and A. Bagtzoglou, “Scalable Localization with Mobility Prediction for Underwater Sensor Networks,” *IEEE Transactions on Mobile Computing*, vol. 10, Mar. 2011, pp. 335–348.

[21] M.A. Ouksel, “The interpolation-based grid file,” *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, New York, NY, USA: ACM, 1985, pp. 20–27.

[22] M.A. Ouksel and O. Mayer, “The Nested Interpolation Based Grid File,” *Proceedings of the 3rd symposium on Mathematical fundamentals of database and knowledge base systems*, New York, NY, USA: Springer-Verlag New York, Inc., 1991, pp. 173–187.

[23] M.A. Ouksel and O. Mayer, “A robust and efficient spatial data structure: the nested interpolation-based grid file,” *Acta Inf.*, vol. 29, Jul. 1992, pp. 335–373.

[24] D. Niculescu and B. Nath, “Ad Hoc Positioning System (APS),” *IN GLOBECOM*, 2001, pp. 2926–2931.

[25] A. Savvides, H. Park, and M.B. Srivastava, “The bits and flops of the n-hop multilateration primitive for node localization problems,” *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, New York, NY, USA: ACM, 2002, pp. 112–121.

[26] K. Langendoen and N. Reijers, “Distributed localization in wireless sensor networks: a quantitative comparison,” *Comput. Netw.*, vol. 43, Nov. 2003, pp. 499–518.

[27] A. Savvides, W. Garber, S. Adlakha, R. Moses, and M.B. Srivastava, “On the error characteristics of multihop node localization in ad-hoc sensor networks,” *Proceedings of the 2nd international conference on Information processing in sensor networks*, Berlin, Heidelberg: Springer-Verlag, 2003, pp. 317–332.

[28] L. Xiao and A. Ouksel, “Scalable self-configuring integration of localization and indexing in wireless ad-hoc sensor networks,” *Proceedings of the 7th International Conference on Mobile Data Management*, 2006.

[29] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, “Vivaldi: a decentralized network coordinate system,” *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA: ACM, 2004, pp. 15–26.

[30] N.B. Priyantha, H. Balakrishnan, E. Demaine, and A.S. Teller, “Anchor-Free Distributed Localization in Sensor Networks,” *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, Los Angeles, California, USA: 2003, pp. 340-341.

[31] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, “TAG: a Tiny AGgregation service for ad-hoc sensor networks,” *SIGOPS Oper. Syst. Rev.*, vol. 36, Dec. 2002, pp. 131–146.

[32] J. Zhao, R. Govindan, and D. Estrin, “Computing Aggregates for Monitoring Wireless Sensor Networks,” *Proceedings of 1st IEEE International Workshop on Sensor Network Protocols and Applications*, 2003, pp. 139-148.

[33] Y. Yao and J. Gehrke, “Query Processing in Sensor Networks,” *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[34] A. Sharaf, J. Beaver, A. Labrinidis, and K. Chrysanthis, “Balancing energy efficiency and quality of aggregate data in sensor networks,” *The VLDB Journal*, vol. 13, Dec. 2004, pp. 384–403.

[35] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, “TinyDB: an acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, vol. 30, Mar. 2005, pp. 122–173.

[36] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, “Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals,” *Data Min. Knowl. Discov.*, vol. 1, 1997, pp. 29-53.

[37] D. Kempe, A. Dobra, and J. Gehrke, “Gossip-Based Computation of Aggregate Information,” *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, Washington, DC, USA: IEEE Computer Society, 2003, pp. 482.

[38] A.G. Dimakis, A.D. Sarwate, and M.J. Wainwright, “Geographic gossip: efficient aggregation for sensor networks,” *Proceedings of the 5th international conference on Information processing in sensor networks*, New York, NY, USA: ACM, 2006, pp. 69–76.

[39] P.B. Gibbons and S. Tirthapura, “Estimating simple functions on the union of data streams,” *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA: ACM, 2001, pp. 281–291.

[40] Z. Bar-Yossef, T.S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, “Counting Distinct Elements in a Data Stream,” *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, London, UK: Springer-Verlag, 2002, pp. 1–10.

[41] S. Ganguly, M. Garofalakis, and R. Rastogi, “Processing set expressions over continuous update streams,” *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, New York, NY, USA: ACM, 2003, pp. 265–276.

[42] S. Nath, P.B. Gibbons, S. Seshan, and Z.R. Anderson, “Synopsis diffusion for robust aggregation in sensor networks,” *Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA: ACM, 2004, pp. 250–262.

[43] H. Yu, “Secure and highly-available aggregation queries in large-scale sensor networks via set sampling,” *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[44] K. Liu, L. Chen, Y. Liu, and M. Li, “Robust and efficient aggregate query processing in wireless sensor networks,” *Mob. Netw. Appl.*, vol. 13, Apr. 2008, pp. 212–227.

[45] S. Kim and S.O. Yang, “Query processing Algorithm in Wireless Sensor Networks,” vol. 4, Apr. 2009.

[46] A.M. Ouksel and D. Lundquist, “Demand-driven publish/subscribe in mobile environments,” *Wireless Networks*, vol. 16, 2010, pp. 2237-2261.

[47] B. Xu, F. Vafaee, and O. Wolfson, “In-network query processing in mobile P2P databases,” *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA: ACM, 2009, pp. 207–216.

[48] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker, “DIFS: A Distributed Index for Features in Sensor Networks,” *Proceedings of First IEEE International Workshop on Sensor Network Protocols and Applications*, 2003.

[49] D. Ganesan, D. Estrin, and J. Heidemann, “Dimensions: why do we need a new data handling architecture for sensor networks?,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, Jan. 2003, pp. 143–148.

[50] D. Ganesan, B. Greenstein, D. Estrin, J. Heidemann, and R. Govindan, “Multi-resolution Storage and Search in Sensor Networks,” *ACM Transactions on Storage (TOS), Volume 1, Issue 3*, 2005.

[51] M. Wu, J. Xu, and X. Tang, “Processing Precision-Constrained Approximate Queries in Wireless Sensor Networks,” *Proceedings of the 7th International Conference on Mobile Data Management*, IEEE Computer Society, 2006, pp. 31.

[52] T. Canli and A.A. Khokhar, “Data Acquisition and Dissemination in Sensor Networks,” *Encyclopedia of Database Systems*, 2009, pp. 548-552.

[53] M. Umer, L. Kulik, and E. Tanin, “Optimizing query processing using selectivity-awareness in Wireless Sensor Networks,” *Computers, Environment and Urban Systems*, vol. 33, 2009, pp. 79-89.

[54] T.Krishnakumar, “Integrated Routing and Query Processing in Wireless Sensor Networks,” *Intl. J. of Applied Engr Research, Dingigul*, vol. 1, 2010.

[55] F. Chen and A.X. Liu, “SafeQ: secure and efficient query processing in sensor networks,” *Proceedings of the 29th conference on Information communications*, Piscataway, NJ, USA: IEEE Press, 2010, pp. 2642–2650.

[56] A. Deshpande, C. Guestrin, and S.R. Madden, “Using Probabilistic Models for Data Management in Acquisitional Environments,” *CIDR*, 2005, pp. 317-328.

[57] J.M. Hellerstein, P.J. Haas, and H.J. Wang, “Online aggregation,” *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA: ACM, 1997, pp. 171–182.

[58] I. Fodor, *A Survey of Dimension Reduction Techniques*, US DOE Office of Scientific and Technical Information, 2002. https://computation.llnl.gov/casc/sapphire/pubs/148494.pdf.

[59] G. Moro, G. Monti, and A.M. Ouksel, “W-Grid: a P2P Self-Organizing Infrastructure for Routing and Data Management in Ad-Hoc Wireless Networks,” *Proceeding of the 2006 International Conference on Pervasive Services (ICPS)*, Lyon, France: 2006.

[60] G. Xing, C. Lu, R. Pless, and Q. Huang, “Impact of Sensing Coverage on Greedy Geographic Routing Algorithms,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, Apr. 2006, pp. 348–360.

[61] B. Karp and H.T. Kung, “GPSR: greedy perimeter stateless routing for wireless networks,” *Proceedings of the 6th annual international conference on Mobile computing and networking*, New York, NY, USA: ACM, 2000, pp. 243–254.

[62] K. Seada, A. Helmy, and R. Govindan, “On the effect of localization errors on geographic face routing in sensor networks,” *Proceedings of the 3rd international symposium on Information processing in sensor networks*, New York, NY, USA: ACM, 2004, pp. 71–80.

[63] K. Fall and K. Varadhan, *Ns-2. The ns-2 manual*. http://www.isi.edu/nsnam/ns/doc/index.html.

[64] H. Tyan and J. Hou, *J-Sim*. www.j-sim.org.

[65] R. Barr, Z.J. Haas, and R. van Renesse, “JiST: an efficient approach to simulation using virtual machines: Research Articles,” *Softw. Pract. Exper.*, vol. 35, May. 2005, pp. 539–576.

[66] L. Xiao, A. Ouksel, and S. Musti, “A simulation system for ad hoc query-ready sensors,” *Proceedings of Grace Hopper Conference for Women in Computing*, 2007.

# VITA

NAME: Lin Xiao

EDUCATION:

B.E., Computer Science, University of Science and Technology of China, 2002

J.D., University of Michigan Law School, Ann Arbor, Michigan, 2010

Ph.D., Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2011

HONORS:

Phi Kappa Phi Honor Society, 2007

Best Paper Award, Grace Hopper Conference for Women in Computing, Orlando, Florida, 2007

PUBLICATIONS:

A. Ouksel and L. Xiao, "Method and system for managing a network of sensors," U.S. Patent No. 7,840,353, 2010.

L. Xiao, A. Ouksel, and S. Musti, "A simulation system for ad hoc query-ready sensors," Best Paper Award, *Proceedings of Grace Hopper Conference for Women in Computing*, 2007.

L. Xiao and A. Ouksel, "Scalable self-configuring integration of localization and indexing in wireless ad-hoc sensor networks," *Proceedings of the 7th International Conference on Mobile Data Management*, 2006.

L. Xiao and A.M. Ouksel, "Tolerance of localization imprecision in efficiently managing mobile sensor databases," *ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, 2005, pp. 25-32.