Distributed Quad-Tree for Spatial Querying in Wireless Sensor Networks

Murat Demirbas, Xuming Lu
Dept of Computer Science and Engineering, University at Buffalo, SUNY, NY 14260
{demirbas, xuminglu}@cse.buffalo.edu

Abstract— **In contrast to the traditional wireless sensor network (WSN) applications that perform only data collection and aggregation, new generation of information processing applications, such as pursuit-evasion games, tracking, evacuation, and disaster relief applications, require in-network information storage and querying. Due to the network dynamics that are typical of WSNs, it is challenging to implement in-network information storage and querying in a resilient, energy-efficient, and distributed manner. To address these challenges, we exploit location information and geometry of the network and present an in-network querying infrastructure, namely distributed quad-tree (DQT) structure. DQT satisfies efficient in-network information storage as well as distance-sensitive querying: the cost of answering a query for an event is at most a constant factor (in our case $2\sqrt{2}$) of the distance "d" to the nearest event in the network. In addition, DQT can handle range query and complex query effectively. DQT construction is local and does not require any communication. Moreover, due to its minimalist infrastructure and stateless nature, DQT shows graceful resilience to the face of failures.**

I. Introduction

Traditionally wireless sensor networks (WSNs) have been treated mostly as data collection and aggregation networks. Examples of such are WSNs deployed for environmental monitoring [18][21] and military surveillance [1][2]. However, as the WSN technology matured, WSNs started to serve more as active information processing tools instead of passive information gathering mechanisms. Examples of these information processing WSNs include pursuer-evader applications [4][19], evacuation applications [3] etc., where mobile entities query the WSN on the spot to learn about their surroundings. Latency and energy-efficiency suffer drastically if these queries are always routed over multiple hops to a centralized base station for resolution. Thus, in-network information storage and querying techniques, such as data centric storage [20] and geometric hash functions [17] have been developed to address these issues.

While in-network querying alleviates the latency and energy-efficiency concerns of information processing WSN applications, certain requirements need to be satisfied by the in-network querying service to be deployable in practice. Firstly, the in-network querying service needs to be distance-sensitive for querying and also efficient for information storage. Distance-sensitivity for querying implies that the cost of answering a query for an event should be at most a constant factor "s" of the distance "d" to the nearest event in the network. Efficient information storage for events implies that the cost of advertising event information is at most a constant factor of the diameter "D" of the network. It is challenging to satisfy both properties simultaneously, since the querying node and the event source are unaware of each other's location and straightforward methods satisfy one of the properties to the extent of violating the other. For example, directed diffusion [13] chooses to optimize the information storage (O(1) cost) to the extent of querying ($O(d^2)$ cost). Combs & needles optimizes querying, O(1), to the extent of information storage $O(D^2)$.

Secondly, to be deployable in practice, the in-network querying service should require minimal infrastructure and its construction should be low cost. In-network querying structures that require costly bottom-up constructions are impractical and error-prone since flooding based constructions are susceptible to severe message losses due to collisions, and may even bring the entire network to a grinding halt. Experimental work found that message loss due to burst of collisions may amount to 50% of total traffic [1][2]. Furthermore, querying structures that employ an elaborate structure may require high maintenance costs due to node failures.

Finally, the in-network querying service should provide graceful resilience to the face of node failures. By graceful resilience, we mean that the performance degradation of querying should be commensurate with the severity of faults. That is, single mote (a WSN node [1]) failure should not impact the performance of querying, the failure of large areas of nodes may impact the performance only proportional to the diameter of the resultant hole in the network and the functionality of querying should be preserved unless the network is partitioned.

Contributions and overview: We present an in-network querying infrastructure, namely Distributed Quad-Tree (DQT) structure, which satisfies all the requirements above, and is suitable for real-world WSN deployments.

DQT maintains a minimalist structure, and in fact, DQT can be considered as stateless. DQT achieves this feat by employing an encoding technique that maps a quad-tree over the deployment area. Just by using the location information at a mote and the coordinates of the top-left and bottom-right corners of the deployment area, our encoding maps a WSN mote to the corresponding level 1 box address. A level 1 box is a smallest partition area in DQT. The addresses of the clusterhead and neighboring clusterheads at each level for a given node are easily derivable arithmetically using the node's DQT address. The implication is that the construction of DQT is local and does not require any communication at all. By exploiting the location information we avoid a costly bottom-up construction.

In our DQT embedding, we choose clusterheads at each level to be the ones closest to the base station at the center of the network rather than the ones closest to the center of the box at that level. Our use of geometry in selecting clusterheads ensures that there are no backward links during the querying and advertise operations. Note that selecting the clusterheads to be the center of each level box results in backward links, and suboptimal paths while going to the clusterheads at higher levels. Our encoding and DQT structure construction is discussed in Section 4.

DQT overlays a quad-tree structure on WSN, and satisfy distance-sensitive in-network querying as well as efficient information storage. The motivation for using quad-trees for in-network querying in WSN comes from the extensive use of quad-trees [7] in centralized algorithms domains especially in the computational geometry area. For example, Quad-tree is used for locating pixels in a 2-D image. Quad-tree partitions the image into recursively four quadrants, where each node (except leaf nodes) has four children. Due to the hierarchical construction, the image can be stored at different layers with more refined resolution at lower layers. Here, for our DQT construction, we overlay a quad-tree in a distributed manner on WSNs. Beside NN query of events, DQT is amenable to being extended to arbitrary and complex queries rather than the binary version "is there an event?" queries. For example, report all the nodes with temperature higher than T1, or report all the nodes within an certain area with temperature in [T1,T2]. Based on different types of range constraints, we further classify the complex range query problems into two categories: range optimization problems and domain optimization problems. These two different types of problems are handled by two specific strategies. The analysis of distance sensitivity of event querying and how DQT is extended to handle the complex range query are discussed in Section 5.

In DQT, since events and intervals information are stored in each level clusterheads, a node exploit local information when the query is introduced or propagated to that node. For example, an event query introduced from the root node may answer the query immediately without communicating to any other nodes simply through searching the local information at various levels. To solve complex range query problems more effiently, we further proposed another query optimization scheme which is called "proactive caching". The principle is to take advantage of relations between parent and children nodes and skip some intermediate querying steps without sacrifice the result integrity. We analyze the optimization scheme in detail in Section 6.

The stateless operation of DQT makes it resilient to the face of node failures and topology changes. To achieve resiliency while routing to clusterheads or neighbors in the structure, DQT maps the DQT address of the destination to the physical coordinates, and leverages on the resilience of a geographic routing scheme (such as GPSR) [15] for delivering the message. Since mote failures do not often lead to failure of a level 1 box, single node failures do not affect the performance of DQT. In the case of failures of motes in an area, GPSR delivers a message addressed to a box in that area to a mote on the boundary of the hole. Since DQT is stateless, the recipient mote easily acts as a proxy on behalf of the intended destination box, and determine the next step in the query or advertise operation by simply plugging the destination box id (instead of its own box id) into the corresponding procedures for the DQT operation. This way, failures of motes in an area degrade the performance of DQT operations proportional to the size of the area. Essentially, the degradation is equal to that of routing stretch in GPSR due to the holes. DQT preserves correct functionality unless the network is partitioned, and even then, functionality is satisfied within each partition. We discuss the resilience of DQT in Section 7.

Our simulation results using ns2 serve as empirical validation of scalability, distance-sensitivity, and resilience of DQT. We present our simulation results in Section 8.

II. RELATED WORK

Centralized querying has been the common mode of querying in WSN. For this mode of operation, the base station acts as the point where the query is introduced and results are gathered. For example, in TinyDB [16], queries are first parsed at the base station and disseminated in a simple binary format into the sensor network to be executed. This centralized structure may not be feasible for distributed and self-organizing sensor networks since: (1) such a base station may not exist, (2) for in-network queries, a query may be introduced from any node in the network and propagating the query to the base station is costly.

Geographic Hash Tables (GHT) [17] gives a simple solution for in-network querying problem: GHT stores and retrieves information by using a geographic hash function on the type of the information. GHT can hash event information far away from the nearby query nodes, and thus violate the distance sensitivity of querying. The average cost of GHT is *D/3* according to [5], where D is the diameter of the network. Although hierarchical version of GHT alleviates this problem, the problem cannot be solved entirely. DQT structure improves over GHT by providing the distance sensitive querying.

To support efficient in-network queries and to store the indexes of data, some sort of hierarchy seems beneficial. Here the idea is to push the query to the higher levels until it is resolved at some level. The query then traverses the subtree to get relevant information. Distance Sensitive Information Brokerage (DSIB) protocol [8] achieves distance-sensitivity in a hierarchically partitioned network by using a push-based approach: an event advertises to neighbors as well as its parents at every level of the hierarchy. DSIB does not require localization information and relies purely on communication topology. To this end, DSIB introduces a costly bottom-up construction and a special purpose routing algorithm. In contrast, DQT assumes localization information and in turn is able to provide an efficient local construction. Using of localization information is not impractical via GPS or other localization techniques. Real-world WSN deployments such as Lites [1] and Exscal [2] already have utilized localization information in their construction. Also DQT relies on the resiliency of the GPSR rather than introducing a routing algorithm.

Distributed Index for Features in Sensor network protocol [9] considers arbitrary and complex queries, and extends traditional binary-tree and quad-tree by allowing multiple parents and multiple roots. DIFS is susceptible to the distance sensitivity problem: a node may have several parents, some of which probably located far away. Moreover, it is costly to construct the DIFS structure, and update operations are also expensive. This is the price DIFS pays to handle arbitrary complex queries about sensor values rather than just binary event information.

III. Model

We assume that the WSN motes sit on a two dimensional plane and their coordinates (x,y) are made available to themselves. We assume a connected network and availability of geographic routing such as greedy perimeter stateless routing (GPSR) [15] or CLDP [14]. There may be coverage holes in the network, but no partitions (i.e., isolated regions). Our analytical results for DQT are proved in Section V in the absence of holes in the network, and in Section VII via simulations we show how they hold up in the presence of holes in the network.

As we describe in the next Section, the network is divided into grid cells while embedding a DQT over the network. A level 1 box in DQT constitutes the smallest cell area in the DQT structure. We assume that all motes inside a level 1 box are within one hop distance. In our terminology, a mote refers to a physical WSN node, while a "node" refers to a virtual DQT node, such a level 1 box.

The cost of querying an event is measured as the number of hops traveled from the querying mote to a mote that holds an advertisement about the event.

IV. DQT Structure and Construction

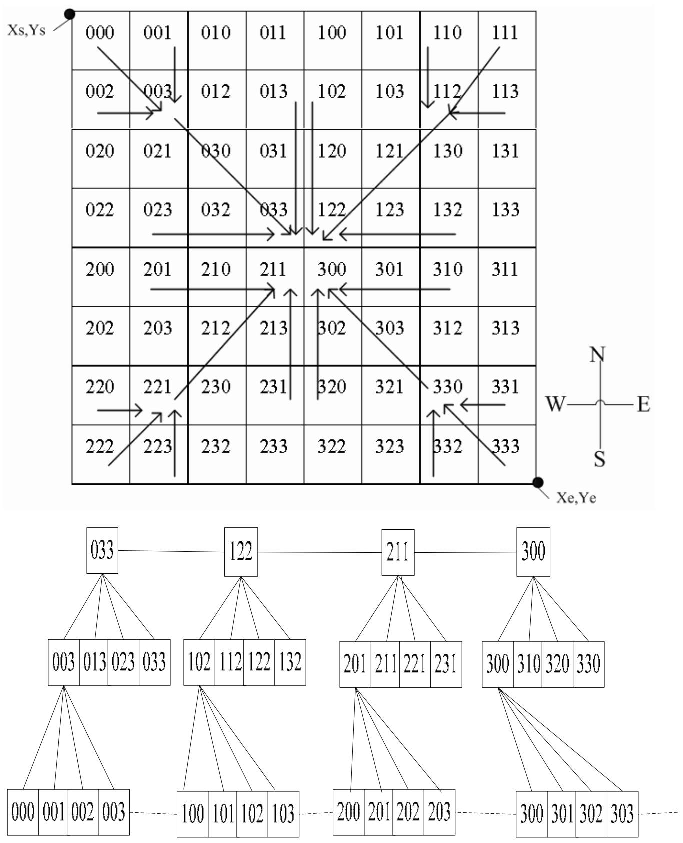


Figure 1, Node addressing and tree structure

For constructing DQT, we employ an encoding trick first presented in [10]. As such, each level 1 box in the structure is assigned an ID which uniquely identifies a region. The length of the ID is equal to the number of levels. We use this addressing scheme to preserve the location information of a node. Due to the way we construct level 1 boxes, this scheme is independent of the number of nodes, but relies on the partition levels. Fig.1 illustrates the addresses of the nodes in a partitioned region with 3 partition levels.

Similar to the centralized quad-tree, DQT is a hierarchical structure. In each level of partition, a node is assigned as clusterhead node of the region. The clusterhead is always its own child in lower levels. The clusterhead at each level partition is statically assigned to be closest node to the geographic center point of the entire network. For example, in level 1 partition, node 003 is selected as clusterhead for 00 region, because it is closer to center than nodes 000, 001 and 002. Similarly, node 033 is selected as level 2 clusterhead, as it is closer to the center than level 2 nodes 003, 013, and 023. The node closest to the center of the entire network in each subpartition is selected as the parent node of that subpartition. The benefit of such a selection is to avoid backward links. For example, in Fig.1, node 000 propagates the query to its root node 033 by first contacting parent node 003, then 003's parent 033. The shortest path is gained since there is no backward link on the querying path. A DQT node may belong to different levels in the hierarchy depending on its location. If a node is a member at level k, it is also a member of level 1. We denote a node p's parent as p.parent and children as p.child. The neighboring nodes are also called siblings, which are denoted as p.sibling.

This structure is quite simple and adapts to multi-dimensional sensor readings, such as (temp, light, humidity), since the construction of DQT is not related to the sensor value. Another difference between DQT and the centralized quad-tree is that DQT does not need a root of the tree. The four nodes in the top level function as the root.

4.1. Mapping from localization to DQT addressing

Each node in DQT can calculate the DQT address of the level 1 partition it resides in from its x,y coordinates easily. Let (x_s,y_s) at NW and (x_e,y_e) at SE be the two endpoints of the area where DQT should be overlayed. Assume DQT have i levels. The area of each level 1 box of partition is $w*l$, where width $w=(y_e-y_s)/2^i$ and length $l=(x_e-x_s)/2^i$. Then DQT address of a node(x,y) can be calculated as:

$$DQT\_addr=\left[\left|\frac{(x-x_s)}{w}\right|(\bmod 2)\right]+\left[\left|\frac{(y-y_s)}{l}\right|(\bmod 2)\right]*2$$

The mapping calculates the X and Y address separately, and then adds them together. We can verify this from Fig. 1, for instance, node ID 033 is obtained by adding 011 and 022, and node ID 332 is obtained by adding 110 and 222. The reason that the second term in the DQT address calculation is multiplied by 2 is because the Y addresses pace by 2 for every increment in DQT addressing scheme. Given this mapping, any node can locally compute its DQT address based on its coordinates (x,y).

Besides the DQT address, each node also maintains its (x,y) coordinate address. This location information is used in GPSR routing in querying and advertising. Since GPSR routing only requires single hop information, which has already been cached as level one neighbors in our structure, it is quite simple to adapt WSN. When the coverage has irregular holes, local optimal path can be reached using right hand rules in GPSR. By using the above encoding trick and assigning DQT addresses for DQT nodes, we can start constructing the DQT structure.

4.2 DQT Local Construction

DQT uses local construction instead of bottom-up construction to reduce communication cost during initial construction. A static and local scheme that uses the address of the box suffices for calculating every level clusterheads and neighbors. Each node may have neighbors at N, S, E, W, NE, NW, SE and SW. The following two methods can be used to find the clusterhead and neighbors at level i.

```
Procedure Cluster_head_Validate (node p,level i)
Switch (p.address(h))
Case 3: //p in SE region
    { If p.address(i) == 0, then return true; else return false}
Case 2: /p in SW region
    { If p.address(i) == 1,then return true, else return false}
Case 1: //p in NE region
    { If p.address(i) == 2, then return true, else return false}
Case 0: // p in NW region
    { If p.address(i) == 3, then return true, else return false}
```

Figure 2 clusterhead validate algorithm

The clusterhead find algorithm discovers the relation of the DQT address to its clusterhead. We find that in NW region of the map, nodes with DQT-address "3" at level i become the clusterheads at the corresponding level. Similarly, in NorthEast partition, nodes with DQT-address "2" at level i become the clusterheads. In Fig.2, p.address(h) is the highest bit of the DQT-address, which determines the region of a node. This algorithm guarantees the clusterheads at each level are closer to the map center than any children (except for itself).

To find the neighbors, we make use of the location information. We use node p and node q to represent the originator and its neighbor. First we use p's location information and increase its coordinates x, y value by a level i box lateral length to find a neighbor node in each direction. If either of x or y value exceeds the range of the map, we ignore that neighbor. For each of these nodes, we find their level i clusterheads. These clusterheads are node p's level i neighbors. For instance, given a node 201 at level 2, we can find its neighbor at north direction by following two steps: (1) Reduce Y value by a level two box length, then we can locate the node 021 through the new (x,y) coordinates; (2). Find the clusterhead for node 021 at level 2, which is 023.

```
Procedure Neighbor_find (node p,level i)
For each direction( N,S,E,W,NE,NW,SE,SW)
{
q.x =p.x + $2^i*l$
q.y= p.y + $2^i*w$
while( p.x &&p.y )
    { Cluster_head_Validate(node q, level i)}
}
```

Figure 3: Neighbor finding algorithm

V. Event query in DQT

Before discussing querying in detail, we discuss how events are indexed in DQT.

5.1 Indexing of event information

000 #	001 #	011 #	012	100	
002 #	003	012 #	013 ##	102	
020 #	021 #	030 #	031	120	
021	023 ##	032	033 ##	122	
022	201	210	211	300	

Figure 4, node 003's sibling structure

In any hierarchical structures as with DQT, some multi-level boundary nodes are far away from each other, while actually they are placed nearby. High latency maybe introduced if the search follows the path of the tree strictly. For example in Fig.1, node 011 and 100 are neighbors. A query from node 011 to node 100 may route to higher level clusterheads such as node 013 and node 033. Our solution is to use sibling links to nearby intermediate nodes. A sibling link is the link between a node and its neighbors in each direction (so each may at most have 8 sibling neighbors). The sibling links only exists between nodes on the same level in the structure. Fig.4 illustrates the vision of an intermediate node 003 in the distributed quad-tree structure. The nodes with "#" are level 1 sibling nodes, the nodes with "##" are level 2 sibling nodes. A node at level i maintains the event information of its cluster, as well as the event information of its neighbors. A node at level i also maintains the interval [Min,Max] information of its cluster for each attribute. Higher level nodes holds wider interval ranges but less detailed information of its subtree. For root nodes also exchanges their interval information so that each root knows the interval of entire structure. This is costless and reasonable since root nodes are adjacent neighbors geographically.

When an event is detected at a level 1 node p, p contacts its immediate parent node at level 1. The parent node updates its record for that child. Node p also contacts it sibling nodes to update their records accordingly. Recursively, the update operation is executed till the top level. This is similar to the information storage scheme discussed in [8] and the sibling links in Stalk [6].

5.2 Nearest Neighbor query

Nearest Neighbor (NN) query is defined as, finding the data object which is closest to the querying object given a set of objects. The classic NN query returns exactly one object as a result. In WSNs, a query can be started at any location. The initiator of a query is the node where the query is entered into the system. The query point is the node for which you want to get NN query result. Query point by default is the same node as initiator of query but it may be specified to be any point in the network.

Our algorithm prevents the propagation of searching to higher levels, if it can be answered locally. Through taking advantage of the spatiality information, both the query efficiency and latency is greatly improved. Our query strategy is to start the query at the query point using local information because the node may belong to multiple levels and therefore hold multi-layer information locally. If no result is obtained, the query is propagated to its parent recursively. At some level the event information is reached, the query is then stopped and returned to the originator.

What if the query point is at another location? That means the initiator of the query and the query point belong to two different nodes. First the query is passed to the query point from the initiator of the query using GPSR routing scheme, and then this querying process is started from the query point. The following results are in the absence of faults. In the simulation section, we talk about the presence of faults.

Theorem 1. *A DQT node at level i stores O (i) information.*

Proof. A node at level i is clusterhead from level 1 to i along the path. The number of neighbor nodes at each level is less than or equal to eight. Therefore the node need 9*i (including one interval record for its subtree) space and stores *O(i)* information.

Theorem 2. *The total space needed for the construction of distributed quad-tree is less than 12*b.*

Proof: According to Theorem 1, level 1 nodes use up a 9*b space internal, where b is the total number of level 1 nodes. Similarly, all level 2 nodes total to a 9*b/4 space usage. Thus, the total space needed for constructing the distributed quad-tree is:

$$9 * (b + \frac{b}{4} + \frac{b}{4^2} + \dots + 1) = 12 * b(1 - \frac{1}{b}) < 12b$$

Theorem 3. *The distance between a level i node and its neighbors is at most $2^i * \sqrt{2}$ hops.*

Proof: According to the partition rule of quad-tree, a level i node is the clusterhead of a $2^i * 2^i$ area. The distance between a level i node and its neighbors is either 2^i (for N,S,E,W neighbors)or $2^i * \sqrt{2}$ (for NE,NW,SE,SW neighbors) depending on the Direction. Since the clusterhead is one of its neighbors at level i, so the distance between a level i node and its clusterhead is also less than $2^i * \sqrt{2}$ hops, which is the diagonal distance of a level i partition.

Theorem 4. *The distance stretch factor s for spatial query in our structure is $2\sqrt{2}$ in worst case. In another words, an event d hops away can be achieved by the querying node within $d * 2\sqrt{2}$ hops.*

Proof: A query from an intermediate level node does not constitute the worst case. The reason is that the clusterhead nodes holds multi-levels information locally and this local cache can be used to answer queries. So, let's consider a query from a bottom level node that reaches a level j clusterhead. We define the query cost as the number of hops from the query point to the node that holds the result.

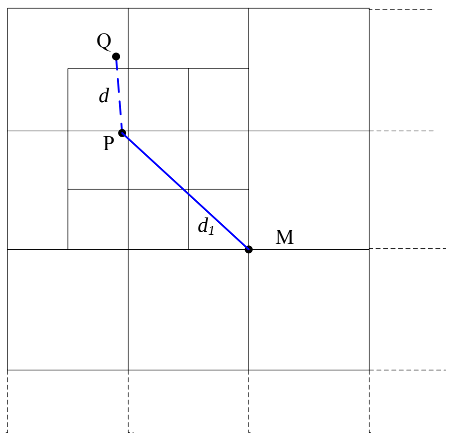


Figure 5. Distance stretch factor s analysis

In Fig.5, d_l is the distance from querying point P to the level node M that the query is propagated, and d is the distance from P to Q. Distance stretch factor s is $s = d_1 / d$.

According theorem 3, the distance from level i-1 node to its parent node (level i) is $2^{i-1} * \sqrt{2}$ hops. Since the backward links are avoided in going-up phase, the total distance from level 1 to level j can be calculated as $(2^1 + 2^2 + .. + 2^{j-1}) * \sqrt{2}$,which is overall $d_1 = \sqrt{2} * 2^j$ hops. Since P and Q are not i-1 level neighbors, the distance $d \geq 2^{j-1}$. The equivalence is true when P and Q are located exactly on the opposite borders of a level i-1 box. Hence:

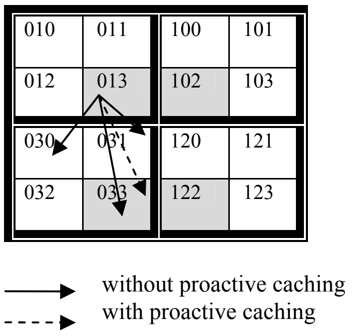
$$s = d_1 / d \leq \sqrt{2} * 2^j / 2^{j-1} = 2\sqrt{2}$$

Based on the result on case 1 and case 2 analyses, we conclude that our structure is distance sensitive with a distance stretch factor $2\sqrt{2}$.

5.3 Event query optimization

As a variation of our event query algorithm, we provide an optimization scheme for event query in the case when the event is not advertised in the neighbors. This means only its parents are published of the events. Thinking of the following query problem: reporting all the event information in the list of IDs(ID1,ID2,..). With brute force solution, this query is sent to all of included nodes. But if there is a query that discovers two sibling nodes which share the same parent, the query could be sent to its parent node directly, since the parent node hold their information. Let's analyze how it can reduce communication costs through proactive caching.

Figure 8 shows an example of parts of the region. The gray nodes (013/102/033/122) are level 1 cluster heads. Suppose the query is introduced through node 013. Since node 030 and node 031 in SW region are neighbors, node 013 may contact the cluster head (node 033) directly, instead of the contact node 030 and node 031 respectively. The table shows that in overall costs, the proactive caching scheme can reduce communication costs (measured as number of hops). In this example, five hops are reduced for south and southeast and east neighbors. The cost in going up phase is reduce by 38% (*T_{up} is 13 hops*).



	L1	L2	Total
without	2	2	4
with	1	0	1

South neighbors of node 013

	L1	L2	Total
without	1	2	3
with	2	0	2

SE neighbors of node 013

Figure 6: proactive caching to reduce communication costs

Let λ^i_j be the querying node's neighbor j at level i. The communication cost C^i_j is measured as the number of round trip hops from the querying node to the target node. The total cost for going up phase without proactive hashing is:

$$T_{up} = \sum_{i=1}^{L} \sum_{j=1}^{N} \lambda^i_j * C^i_j$$

where *L* is the highest level that the query is propagated and *N* is the maximum number of sibling neighbors at level i. We have *L <= O(logn)* and *N <= 8.* The cost is not only rely on the query itself (Which level the query may arrive), but also depends on the location of the query point. For example, the nodes at boundary area have fewer neighbors at each level than internal nodes. The nodes close to center point may have fewer costs for some neighbors.

The optimization scheme mainly optimized the going up phase. We denote U_i *as the* nodes ($U_i \in [0, N)$) in level i neighbors that also belongs to level i+1 cluster heads, and have additional neighbors belongs to children; denote V_i as the nodes in level i neighbors that also belongs to level i+1 cluster heads, but do not contain any children in level i neighbors. We have U_i+V_i *<=8.* For example node 030 in figure 1, the level-2 neighbor at SE is node 300, which do not have other 030's neighbor nodes whose parent is node 300. So node 030 belongs to V_i and node 211 belongs to U_i.. So the reduced cost *(R_c)* in our scheme is:

$$R_c = \sum_{i=1}^{L} \sum_{j=1}^{2U_i} \lambda^i_j * C^i_j + \sum_{i=1}^{L} \sum_{j}^{V_i} \lambda^i_j * C^i_j$$

We find from the equation that the reduced cost is more sensitive to the first part: that is the nodes belong to U_i group make more contributions to reduce the communication cost in our algorithm. *The best case is when* V_i=0, all neighbors belong to U_i group, the cost reduction is $\sum_{i=1}^{L} \sum_{j=1}^{2U_i} \lambda^i_j * C^i_j$; whereas the worst case is when *Ui* =0, all the neighbors belong to V_i group, the cost reduction is $\sum_{i=1}^{L}\sum_{j}^{V_i}\lambda_j^i * C_j^i$. The best case happens when the level i partition is an internal box, with no neighbor crossing the highest level partition borders, while worst case happens if that box contains the center point.

VI. Complex Range Query

The purpose of range query is to find a group of nodes in a given query range. The results are unknown, it may involve all the nodes if the range covers, or may return empty if the query is out of the range limitation. Very similarly, the query can be started from any nodes and is executed in a distributed manner. Following are some of the range query examples in WSN: What are the nodes whose temperature is great than T1 within a certain area? Reporting or counting the sensor nodes within the temperature range [T1,T2] and pressure range [P1,P2] ? In general cases, the queries are associated with geographic constrains, e.g, report the nodes with temperature greater than T1 within the rectangle area enclosed by [x1,x2] and [y1,y2]. If there is no explicit range constrains in the query, e.g, reporting the nodes with the temperature in [T1,T2] and pressure in [P1,P2], by default the query range is the entire coverage area. This is considered as a special case in complex range query. In this section, we propose a generalized algorithm to resolve the complex range query problems.

In complex range queries, the results are a set of ordered nodes in compliance to the measurement. Another difference is range query contain both going-up and going-down phase, while only going-up phase in necessary for event query. Our algorithm reduces the complexity and space in [9] in that instead of storing the T(p) information for all the siblings from p to the root, this algorithm only stores the information of the sibling for current level, except some nodes that belongs several levels. Each sensor node p at level i stores the neighbor information at current level, as well as the T(p) information for it's subtree. If the entry node itself is the cluster head for multi-levels, even more efficiency is achieved through local caching, because the locally saved high level information can be used to answer query or prune branches. This is because a level 3 node is also a level 2 and level 1 node, and a level 2 node is also a level 1 node. In the case a query is sent to a level 2 node from child, if it is also a level 3 node, then the query is pushed to level 3 without any communication cost.

Our strategy to resolve complex range queries is to make use of the geometric information. Suppose query range is enclosed area with points (x1,y1) and (x2,y2). First we calculate the span of each direction S_h and S_v:

$$S_h = \log_2(\left|\frac{x2 - x1}{l}\right|) + 1$$

$$S_v = \log_2(\left|\frac{y2 - y1}{w}\right| + 1$$

where *w* and *l* is calculated in section 4.1. Then we are confident about the following facts: horizontal constrain is within two level S_h neighbors, and the vertical constrain is within two level S_v neighbors. Then, the constrained area lies within two level MAX(S_h, S_v) neighbors overall. The query is then directly propagated to a level MAX(S_h,S_v) node in the constrained range. For example, if MAX(S_h,S_v)==1, the query region lies within two level 1 neighbors. We need to proof this query strategy does not leak any results. Since we choose MAX(S_h,S_v) as the level of nodes to query, we only need to guarantee that in each direction our strategy contains all the possible nodes. It is simple to verify that for unit length *l* and width *w* , they can only lie within *2*l* and *2*w*(which is two level 1 neighbors) distance in each corresponding directions; similarly, for each *2*l* and *2*w*, they lie within *2*(2*l)* and *2*(2*w)*, which is two level 2 neighbors; and so on and so forth…so this scheme is guaranteed of correctness. Our algorithm to solve complex range query problems is show in Figure 6 and Figure 7. The general idea is to find the least level pair of neighbors which can cover range constraints and send the query to that clusterhead using GPSR protocol. This is a top down scheme. It is efficient because we simplified the going up phase and reduced cost, it is correctness

For the special case as we mentioned before, the query is sent to the root node since the root nodes hold the interval information of the entire network. The result of MAX(S_h,S_v) equals to the number of levels in the structure. If any of the query range is beyond the recorded interval, then empty result is returned immediately. Otherwise, select one of requirement with tight most constraint attributes as the primary pruning key. Then this query is pushed down to each child with necessities pruning of subtrees.

```
RangeQuery_topdown (Node q, Nodeset p, Metric m) {
While(q.sibling){
If ($m \in T(q)$)
then
   { initialize the partition list of p=q.children
      for each p=p.child in the list
         {  If($m \in T(p)$)
                        if p is a leaf node
            {then add p to the nodeset RQp}
                      else (add p to the nodeset RQp and
                           send query to its children)
            else prune the subtree
         }
      }
      Return RQp;
}
else
   return EMPTY
}
```

Figure **7:** Top down procedure for clusterhead nodes.

```
RangeQuery (Node q, Nodeset p, Metric m) {
calculate i = Max(S_h,S_v);
Cluster_head_validate(node q', level i);
send query q';
Start RangeQuery_domain procedure in the current node;
}
```

Figure **8:** Algorithm for complex range queries.

Why we use top-down querying scheme to answer range query? The reason is that any value can be located anywhere in theory. We cannot filter out any nodes unless the range is specified explicitly in the query. But if some model of sensor data is available, other optimization schemes can be applied such as proactive caching and bottom-up approaches, which are left for future work.

VII. FAULT TOLERANCE

DQT is fault tolerant due to several aspects.

First, any leaf mote failure does not cause any update operation and structure change. For a dense sensor network, each level 1 partition contains several nodes. All nodes in the same partition share a common DQT ID. DQT structure is stateless, which means the nodes do not need to maintain a state of its own. The neighbors are computed in initial construction phase through local computing, which not only reduces communication cost but also improves the system reliability.

Second, DQT can handle coverage holes nicely. Only if all the motes inside a level 1 partition fail (although this is unlikely to happen) a hole may be formed in DQT. Using the right hand rule, GPSR re-routes the information to the closest node to the target node, which we call proxy node. The proxy node pretends to be the target node and find its neighbors through local computation. Then it follows the event or range searching algorithm we discussed in previous sections. The structure is robust once constructed. Shape change of a hole only changes the selection of proxy nodes, and has little influence on other nodes.

In event query scenarios, failure of nodes may cause the following two cases. Case 1: Failures happen before the event advertisement. When a target node of event advertising fails, the event is published to proxy node by default, which in theory is the closest node to the failure node. The failure of advertising destination node will not affect the query result in theory, since it can reach the proxy node. Case 2: The event has already been published in the structure before the failure happens. In this case, the event is still reachable unless all the nodes along the querying path were dead. When a node with event advertisement dies, queries to this node are passed to its parent node. We will further discuss the performance of each case in simulation part.

VIII. SIMULATION

We investigate the performance characteristic of DQT using the ns-2 wireless network simulator. Our simulations mainly focus on two aspects: stretch factor and fault tolerance. 256 nodes are simulated in our experiment, which are uniformly distributed in a 2-dimensional square of 3200x3200. The distance between each node is 200m, while the transmission range is set to be 250m. Therefore the average degree is about 4 in the field except some border nodes. That is, not all the level 1 neighbors are reachable via single hop. The geographic location of each node is available, and is used to construct the DQT structure in initial phase. The height of the DQT tree is 4, with 4 roots at the top level. The cost of querying an event is measured as the number of hops from the querying node to the node that holds an advertisement about the event.

Our experiments focus on node-level behavior instead of mote-level behavior. Recall that each node represents a level 1 box in the map. As a result, a node failure in our experiment means all the motes in an area of the corresponding level 1 box failed. Currently our simulation only handles the event querying. The performance for complex range query is left for future work.

8.1. Stretch factor in the absence of faults

We have proved in Theorem 4 that the stretch factor in worst case is $2\sqrt{2}$. We calculate the average distance stretch factor through 100 runs of each experiment. In each round, a query/sink node pair is randomly chosen. We use two measurements s and s', where s is the ratio of the DQT querying cost to the distance between the query and the event node and s' is the ratio of the DQT querying cost to the GPSR cost. The value of s show the ratio to ideal cost, whereas s' ratio of routing a message from the querying point to the event. We found the average s is around 0.6 and s' is around 0.5 in absence of faults. The reason that s is much smaller than $2\sqrt{2}$ is that the worst case scenarios only occupy a small percentage of the total. Our scheme has a considerably smaller stretch factor compared to the DSIB scheme, which has an average value 0.9~1.

Fig.9 illustrates the average stretch factor s and s′, as well as their standard deviations where the event and querying pairs are randomly selected with varied distance between the query node and event node. For nearby pairs, the s and s′ tend to be close to 1, since either GPSR or DQT makes little difference. The average stretch factor s decreases with the increase of the distance of query/event pairs. But we also find, that when the distance is close to the diameter of the map (such as to 14/16 etc), s and s′ slightly increase again. We call this phenomenon border effect. The reason is that when the pair of nodes approach the borders of the maps, they are less likely to be connected through their common neighbors.

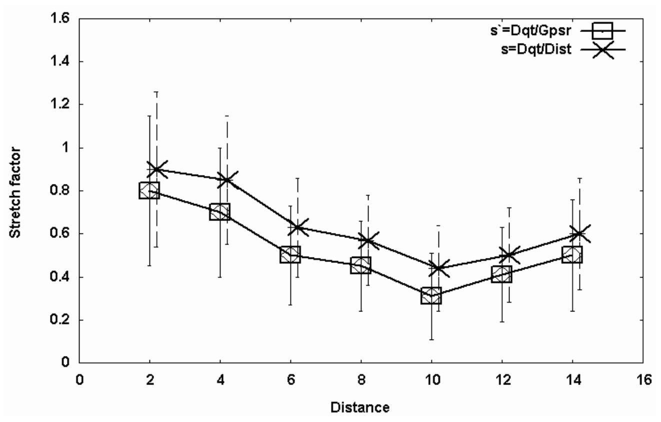


Figure 9: Stretch factor s and s´ with varied query distance

8.2 Fault tolerance

DQT is robust in that a single mote failure will not change the structure and the query operation. To evaluate the performance of DQT, node failure in the structure is experimented. We use the same topology and setting in our experiment. We randomly remove a certain percentage of nodes, namely from 2% to 20%.

We first experiment on the DQT query success rate with node failure. Note that, for case 1, the event still publishes in proxy node when the destination node fails. Theoretically there is no failure for querying, unless GPSR fails to forward along the path or the network is isolated. In case 2, the query is extended to the parent node when a node with event advertisement fails. A query may fail when all event nodes along the querying path fail. Fig.10 shows that for case 1, when query success rate can drop down to 95% when 20% of nodes fail. However, for case2, the DQT scheme itself may fail besides the GPSR routing failure. The failure rate goes up to

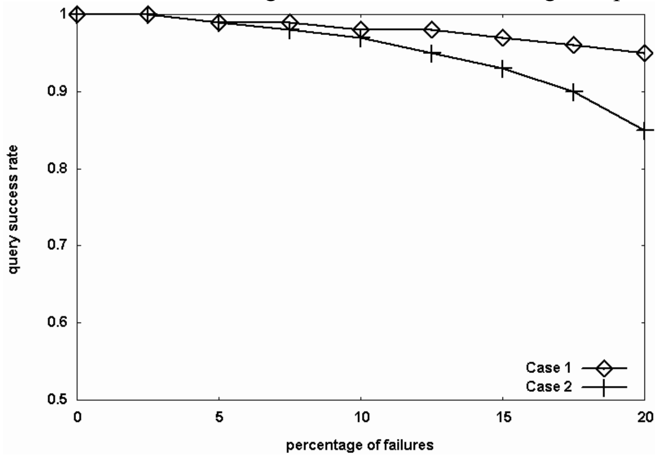


Figure 10: DQT success rate for case 1 and case 2

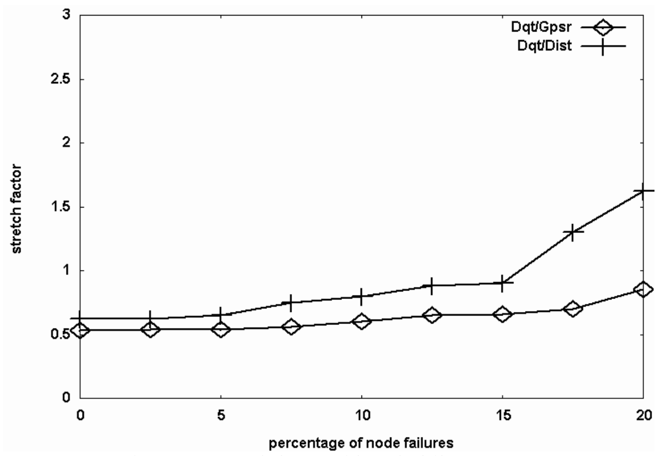


Figure 11: Stretch factor with node failure: Case 1

15% in case 2. The result may vary in real environment due to the increase of GPSR failure and link asymmetry. Increasing the degree of nodes or node density is helpful in improving the query success rate.

From the stretch factor point of view, case 2 is also worse than case 1. Fig.11 and Fig.12 illustrate the stretch factor with varied possibilities of node failure for case 1 and case 2. In both cases, DQT works fairly well within 10% failure of nodes. With the increase in the failure rate, the value s get worse quickly for both cases, because the query circumvents the holes (due to failure), which increases the cost of searching rapidly. Case 1 performs better than case 2 because in the occurrence of holes, case 2 circumvents the hole and query its parent, while in case 1, the event is achievable through a proxy node. The s′ remains relatively small since the same overhead applies for GPSR to overcome the coverage holes. The results also indicate that the degradation of performance is smooth overall.

Finally, for case 2, we plot the trends of s and s’ with respect to query distance in Fig.13. Obviously, the failure of nodes affects the overall performance of the DQT. When the failure rate increase, both s and s’ will increase for each distance. The s and s’ is more closely related in the absence of failure or when the failure rate is small.

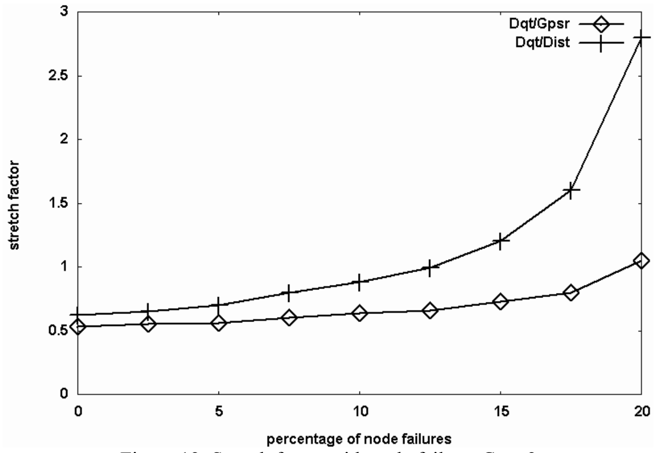


Figure 12: Stretch factor with node failure: Case 2

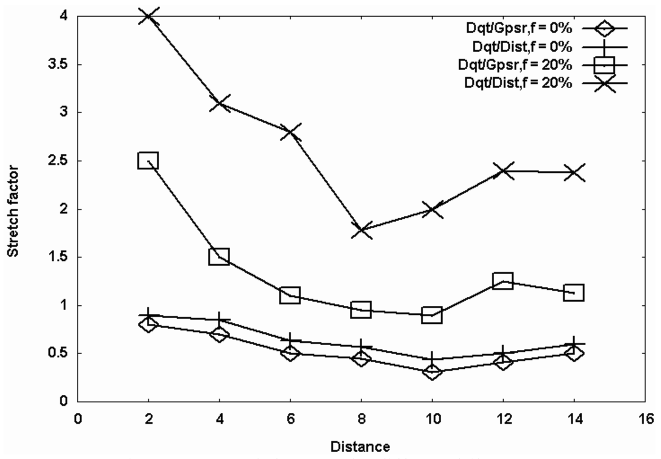


Figure 13: Stretch factor under different failure rate

IX. Conclusion

We presented an in-network querying infrastructure, namely distributed quad-tree (DQT) structure, suitable for use in real-world WSN deployments. DQT satisfies distance-sensitive querying as well as efficient information storage in network. DQT construction is local and does not require any communication. Moreover, due to its minimalist infrastructure and stateless nature, DQT shows graceful resilience to the face of node failures.

DQT is amenable to being extended to arbitrary and complex queries rather than the binary version “is there an event?” queries we presented here. Since the queries are arbitrary, the information advertisement cannot anticipate all queries, and only a summary of sensor data is stored for energy-efficiency purposes. As such, for resolution of queries there may be several matching options that need to be explored but may not satisfy the query and may result in back-tracking. In this case several query optimizations are possible to improve the performance. We present a DQT solution for answering complex range querying and a “proactive caching” optimization scheme further to improve query efficiency.

The stateless nature of DQT makes it resilient to topology changes. In fact, it may be possible to extend DQT to provide a location service for mobile ad hoc networks. The idea is to retry a query until it catches up with the mobile target. Even though a target node may move during the query execution and leads to a miss, the query when invoked from this new location closer to the target node will have a better chance to catch up to the target node due to the distance-sensitivity property in DQT.

References.

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, et al., A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking, Computer Networks, Vol. 46, Issue 5, pp. 605-634, Dec. 5, 2004.

[2] A. Arora, R. Ramnath, E. Ertin, and P. S. et. al., Exscal: Elements of an extreme scale wireless sensor network, in 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2005.

[3] C.K.Constantine, Sensor Networks Applied to the Problem of Building Evacuation: An Evaluation in Simulation, Proceedings of the 15th IST Mobile and Wireless Summit, June 2006, Mykonos, Greece.

[4] M. Demirbas, A. Arora, and M. Gouda, Pursuer - Evader Tracking in Sensor Networks. Sensor Network Operations, chp.9, IEEE Press, May 2006.

[5] M. Demirbas, A. Arora, and V. Kulathumani. Glance: A Lightweight Querying Service for Wireless Sensor Networks. OPODIS'06 December 2006.

[6] M. Demirbas, A. Arora, T. Nolte, and N. Lynch. A Hierarchy-based Fault-local Stabilizing Algorithm for Tracking in Sensor Networks. 8th International Conference on Principles of Distributed Systems (OPODIS), France, December 2004.

[7] R. Finkel and J.L. Bentley , Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica ,1974,4 (1): 1-9.

[8] S. Funke, L. J. Guibas, A. Nguyen, and Y. Wang. Distancesensitive routing and information brokerage in sensor networks. In *DOCSS*, 2006.

[9] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. Difs: A distributed index for features in sensor networks. First IEEE Ineternational Workshop on Sensor Network Protocols and Applications, May 2003.

[10] I. Gargantini. An effective way to represent quad-trees. Commun. ACM, 5(12):905--910, 1982.

[11] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, Fractionally Cascaded Information in a Sensor Network, Proc. of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN'04), 311-319, April, 2004.

[12] J.Hightower, G.Borriello,"Location systems for ubiquitous computing." IEEE Computer, Vol. 34, No. 8, August 2001 pp 57-66

[13] C. Intanagonwiwat, R.Govindan ,D.Estrin, Directed diffusion: a scalable and robust communication paradigm for sensor networks, Proceedings of the 6th annual international conference on Mobile computing and networking, p.56-67, August 06-11, 2000, Boston, United States.

[14] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In Proceedings of NSDI 2005, pages 217.230, May 2005.

[15] B. Karp and H. T. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In MobiCom: Proceedings of the 6th annual international conference on Mobile computing and networking, pages 243–254, 2000.

[16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, San Diego, June 2003.

[17] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: a geographic hash table for datacentric storage. In WSNA ’02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications, pages 78–87, 2002.

[18] R. Szewczyk, A. Mainwaring, J. Polastre and D. Culler, An Analysis of a Large Scale Habitat Monitoring Application, Sensys 2004.

[19] L. Schenato, S. Oh, and S. Sastry, Swarm coordination for pursuit evasion games using sensor networks, in Proc. of the International Conference on Robotics and Automation, Barcelona, Spain, 2005.

[20] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan and D. Estrin, Data-centric Storage in Sensornets, in Workshop Record of the First Workshop on Hot Topics in Networks (HotNets-I), October 2002

[21] G.Tolle, J.Polastre, et.al, A Macroscope in the Redwoods, In Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys), November 2-4, 2005.