

Reliable Hierarchical Data Storage in Sensor Networks

Song Lin, Benjamin Arai, Dimitrios Gunopulos
Department of Computer Science & Engineering
University of California, Riverside
Riverside, CA, USA 92521
{slin, barai, dg}@cs.ucr.edu

Abstract

The ability to provide reliable in-network storage while balancing the energy consumption of individual sensors is a primary concern when deploying a sensor network. The main concern with data-centric storage in sensor networks is the ability to provide reliable and load balanced storage. Energy and wireless range constraints make centralized approaches for storage impractical, and in-network data-centric solutions can be used to reduce the number of messages sent over the network. However, these solutions quickly become expensive when combined with fault-tolerance, load balancing and routing. In this paper, we present a novel data-centric storage and query routing mechanism for sensor networks. The routing mechanism is constructed upon the neighborhood information of individual sensors and is completely independent of geographical information. Our data resilient algorithm is capable of recovering from multiple simultaneous failures in the network while adaptively adjusting the load distribution of the newly generated sensor data. Comprehensive experiments on both real-world and synthetic data sets indicate that our approach is more effective and efficient than the previously proposed solutions.

1 Introduction

Recent developments in sensor technologies have made it possible to store large amounts of data within individual sensors [11]. This ability to delay the transmission of data can provide benefits for both accuracy and energy efficiency. For example, we can store sensor readings at individual sensors and selectively transmit only the fraction of the data that is of interest to the user. To store data and efficiently answer queries in a sensor network, there are several distinct limitations that must be addressed. Among these limitations, energy is usually the primary concern when deploying the sensor network as the sensor devices are either

powered via a limited power source (such as batteries), or renewable sources (such as solar energy) that allow continual use of energy but at a limited rate of consumption [21].

A possible way to solve this energy limitation problem is reducing the network traffic. Many sensor network algorithms have focused on techniques for minimizing the number of messages sent over the network via, lossy (approximation) or other routing focused solutions. Another possible solution is the distributed storage scheme which reduces the number of messages sent over the network by storing messages with little impact at individual sensors for later analysis. These stored readings can be later analyzed to determine if the readings merit transmission to the query sensor or should be disregarded. Distributed systems such as Peer-to-Peer [16, 22, 25] have offered data-centric storage solutions but cannot be directly applied to sensor networks because of the large network traffic associated with the data routing in the overlay network. On the other hand, recent data-centric storage algorithms in sensor networks usually require geographical location, which is not always available (e.g. if the sensors are not equipped with GPS locator, or if the sensing area is covered by some large obstacles which stops GPS system from receiving signals from the satellites). We propose to construct a hierarchical topology in the sensor network based only on the neighborhood information at individual sensors for efficient query routing and traffic reduction.

In addition to reducing network transmissions we are also concerned with the reliability of the data stored within the sensors. We address this by implementing a reliable data storage architecture where one or more sensors can become inaccessible while still upholding the data integrity of the network. The ability to efficiently balance queries in such a way that a single sensor does not become a bottleneck or overloaded is pivotal in ensuring the maximum network lifetime and quick query response. We do this by employing a unique load balancing algorithm in such a way that the data at high-traffic sensors is offloaded to two or more

sensors to help distribute the load among other less busy sensors.

In combination of efficient routing, reliable storage, and query load balancing, we propose a distributed storage system for storing data in a distributed manor across all sensors in a sensor network. Our paper makes the following contributions:

- We present a hierarchical storage system where all sensors cooperate in storing data into a single database.
- We propose a resilient data transmission mechanism to work against link failures during the routing.
- We present a data reliability algorithm for recovering data from sensor failures with a self-mending mechanism.
- We propose an efficient load balancing algorithm to equalize the amount of data stored in individual sensors in such a way that energy expended at individual sensors is balanced.

The rest of this paper is organized as follows: In section 2 we present the related work. In section 3 we present the problem formulation, in section 4 and 5 we present our Interval Tree algorithm for indexing both one-dimensional and multidimensional sensor data in the sensor networks, in section 6 we provide our experimental results and we conclude the paper in section 7.

2 Related Work

Our paper is closely related to several problems in the area of distributed systems in sensor networks namely: In-Network storage systems, Reliable storage systems and Query load balancing.

In recent years a large number of approaches have been presented targeting the problem of storing data generated inside a sensor network. There are three major strategies to solve the data storage problem in sensor networks: central storage, local storage and data-centric storage. The first approach is to send the data readings in the sensors to a centralized server or base station where it is stored and processed during the query evaluation [3, 9, 24, 19, 13]. This strategy is suitable for streaming data applications or in scenarios where most of the data generated by the sensors will be used by the query processor. However it is proven to be too costly in communication overhead when the user is only interested in a small fraction of the sensor data. On the other extreme, the data can be stored locally in the sensor from which the data was obtained without any data transmission. Whenever a query is issued by some user, the query has to be flooded to every sensor in the network and each sensor transmits back the qualified local results for the query. This

is also an expensive approach when a small fraction of the sensors have qualified data for the query.

The third approach, data-centric storage, organizes the sensor data into the network with a mapping function. The data-centric approach utilizes a mapping function which maps every data object generated in the network to a sensor called owner based on some attributes in the data object [20, 15, 8, 2]. The owner is responsible for the storage of the data object and processes locally queries referencing this object. When a user wishes to query the network, he can send the query only to the owner node responsible for the data relevant to the query through some efficient geographic routing mechanism (such as [18], [17]) without flooding the query in the network. The different data-centric systems presented in the literature differ mainly in the mapping function used which could be a hash function [20, 15] or a tree like structure [8, 2]. A common feature for most of them is that they all require knowledge of the geographic location of the sensors which is not always possible (e.g. if the sensors are not equipped with GPS locator, or if the sensors area is covered by some large obstacles which stops GPS system from receiving signals from the satellites).

To the best of our knowledge, *GEM* [12] is the only data-centric routing and storage system without geographical information of the sensors. In *GEM*, a labeled graph is computed and embedded into the original network topology. The labels assigned to the sensors allow messages to be efficiently routed through the network, while each node only needs to know the labels of its neighbors. *GEM* utilizes only the leaf nodes to index and store the sensor data, which wastes the storage capacity of the internal nodes in the graph. In addition, *GEM* does not provide any recovery mechanism for data loss due to node failures and the data size maintained at different sensors cannot be balanced dynamically according to the distribution of the sensor data.

Surprisingly limited research has been done in the area of reliable data storage for sensor networks. In order to make the DCS systems reliable, several works (like [20, 15]) send a special refresh message from the owner node in the network to all nodes which have generated objects stored at the owner. A GPSR [7] routing protocol is then utilized to return these refresh messages to the owner node with a network perimeter attached. If it is discovered that there is a new node closer to this location than the original owner then the new node will become the owner of the object and will start transmitting refresh messages. This process, however, does not protect the network against data loss due to node failures as the data kept at the failure node will be lost. In [1] the authors propose a Resilient DCS system to achieve scalability and resilience by replicating objects in strategic locations in the network. The idea is to store the object at R different replica nodes generated by a hash function. The replica nodes keep exchanging information in order to get a

consistent overview of the object generated in the network. This approach, though effective against failures, requires a global view of the network topology along with the position of each sensor, and thus is too expensive or impractical for many sensor network applications.

Existing sensor network storage systems like Cougar [3] or ThinyDB [10] are oriented toward query processing within the limits of the sensor network where the results of the query are delivered to a user workstation outside of the network. The prevailing approach for load-balancing in a sensor environment is through creation of balanced trees which cover the nodes inside the sensor network [5] [4]. However, the construction of the balanced tree is centralized which means that the trees have to be constructed in a base station and remain static during the network life time. Kakiuchi [6] proposed an algorithm to modulate the routing tree when it is discovered that the load of some sensor is exceeding a predefined threshold. However any adjustment in the tree introduces the changes of the whole network topology which creates large amounts of network traffic.

3 Problem Formulation

Let \mathbf{S} denote a sensor network of n sensors measuring some environmental quantities (e.g. temperature, pressure, lighting, movement, etc) of some area G . Each sensor generates a record r every ϵ seconds. In order to efficiently answer queries on the sensor data, we segment the data domain D (e.g. $0F - 150F$ for temperature) into several non-overlapping intervals (e.g. $0F - 10F, 10F - 20F, \dots$), and assign different intervals to different sensors. When a new data record \bar{r} is generated at some sensor S_i , it will find the interval $I_{\bar{r}}$ that \bar{r} belongs to, and then be transmitted to the sensor S_j that is in charge of the interval $I_{\bar{r}}$. Similarly, when a query is initiated at some sensor S_k , the query message will be routed to the sensors that maintain the qualified data for the query and the query results will be transmitted back to the query sensor. Our problem can be formulated as follows,

Given a sensor network of n sensors where each sensor S_i has a neighbor list indicating the set of sensors that S_i can communicate directly. The goal is to segment the domain D of the sensor data into several non-overlapping intervals and assign different intervals to different sensors, such that we can efficiently answer the Equality Queries and Range Queries from any sensor in the network.

Definition 1. *Equality Query:* a one dimensional query $Q(v, a)$ in which the field value v of the record r is equivalent to value a .

For example the query $Q(\text{temperature}, 100F)$ can be used to find the data records with the temperature as $100F$.

Definition 2. *Range Query:* a one dimensional query

$Q(v, lb, ub)$ in which the field value v of the record r is between the lower bound lb and upper bound ub . The *Equality Query* is a special case of the *Range Query* $Q(t, lb, ub)$ in which $lb = ub$.

For example the query $Q(\text{temperature}, 80F, 90F)$ can be used to find the data records with the temperature between $80F$ and $90F$.

Definition 3. *Multi-dimensional Range Query:* a multi-dimensional query $Q(v_1, v_2, \dots, v_n, V_{1lb}, V_{1ub}, V_{2lb}, V_{2ub}, \dots, V_{nlb}, V_{nub})$ in which the attributes v_1, v_2, \dots, v_n are in the multi-dimensional query range ($[V_{1lb}, V_{1ub}], [V_{2lb}, V_{2ub}], \dots, [V_{nlb}, V_{nub}]$) (i.e. $V_{jlb} \leq v_j \leq V_{jub}$ for $1 \leq j \leq n$).

Evaluating the above queries efficiently requires a reliable storage and load balancing data-centric storage algorithm, which can be segmented into 4 sub-problems as follows,

- *Interval Distribution:* How to assign different intervals to different sensors for query routing?
- *Resilient Routing:* How to set up a backup routing if some communication link fails?
- *Reliable Storage:* How to provide and distribute some redundancy for the sensor data so that we can recover the lost data stored at the failure sensor effectively?
- *Load Balancing:* How to change the interval lengths at different sensors adaptively in such a way that the sizes of the data stored at different sensors are similar?

4 Indexing the Sensor Data

In this section, we present the *Interval Tree* algorithm, a reliable and load balancing data-centric storage algorithm for sensor networks. More specifically, we show how our algorithm deal with the problems related to data-centric storage in sensor networks, namely *Interval Distribution*, *Resilient Routing*, *Reliable Storage* and *Load Balancing*.

4.1 Tree Construction

We use a tree-like structure, *Interval Tree*, as the network topology for both data storage and in-network routing in sensor networks. Similar to TAG [9] or COUGAR [23], the algorithm constructs a routing tree that covers all the sensors in the network. It starts out by assigning a random sensor in the network as the *Root* and then the *Root* broadcasts the *tree construction message* to its neighbors asking more sensors to organize into the routing tree. The *tree construction message* includes the node id of the sender, and the level of the sender (i.e. the distance from the sender to the *Root*). Any sensor without an assigned level, once receiving a *tree construction message* from a sensor S of level l ,

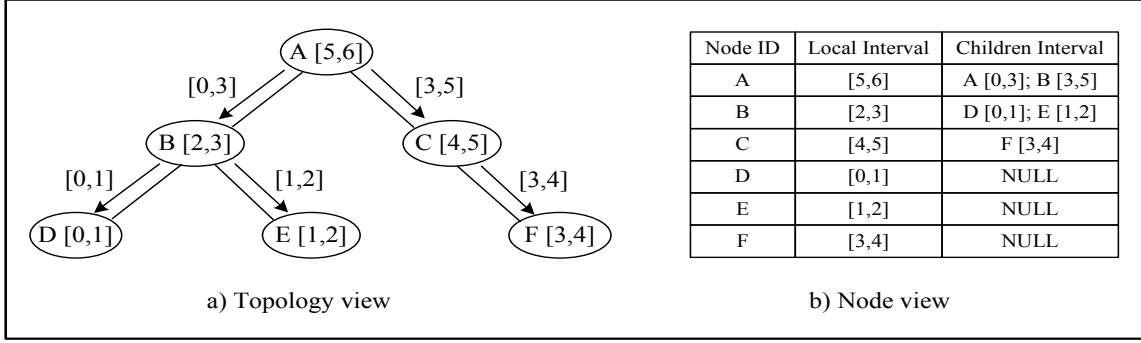


Figure 1. The *Interval Tree*

confirms S as its parent, assigns itself as level $l + 1$ and then broadcasts the *tree construction message* to its neighbors. The *tree construction message* is flooded into the network until all the sensors in the network have been assigned a level and a parent. With the level and parent-children information for each sensor node, a routing tree topology is constructed for the sensor network. It is easy to see that in a dense sensor network, given the communication range of sensors (r) and the longest distance between sensors (D), the height L of the tree can be bounded by $L \leq D/r$.

After the tree construction, a counting process is performed to guide the distribution of data intervals to all the sensors in the network. Starting from the leaf nodes (having no children nodes) to the *Root*, each node computes the size of the sub-tree rooted from it (self-count) and the size of the sub-tree rooted from each of its direct children (child-count). Take Figure 1 as an example, sensor A has two direct children B , C which have 2 and 1 direct children separately (which are at the bottom level of the tree). After the counting process, sensor A knows that there are 6 nodes total in the subtree rooted from A (i.e. $\text{self-count}(A) = 6$). In addition, A also knows that there are 3 nodes in the subtree rooted from B ($\text{child-count}(B) = \text{self-count}(B) = 3$), and 2 nodes in the sub-tree rooted from C ($\text{child-count}(C) = \text{self-count}(C) = 2$). This counting information is used to distribute the data intervals to different sensors as we will describe in the next subsection.

4.2 Interval Distribution

Given the domain of the data records $D = [LB, UB]$ (e.g. $0F - 150F$ for temperature records) and n sensors in the network, we evenly segment D into n non-overlapping intervals $I_j = [LB + \frac{UB-LB}{n} \times j, LB + \frac{UB-LB}{n} \times (j+1)]$ ($0 \leq j \leq n - 1$). Starting from the *Root*, each node locally indexes the last one of the given intervals (Local Interval) and assigns the other intervals to its children proportionally to the child-counts at different children (Children Interval). We keep assigning intervals level-by-level from the *Root* towards the leaf nodes of the tree until all the sensors have

been assigned their corresponding intervals. Let us consider the example shown in Figure 1 again, we want to assign 6 intervals (i.e. $[0,1]$, $[1,2]$, $[2,3]$, $[3,4]$, $[4,5]$, $[5,6]$) to sensors. Starting from the root, which is given the data domain $[0,6]$ of 6 intervals, A assigns the last interval to itself $[5,6]$ and assigns the first 3 intervals ($[0,3]$) to its left child B (as $\text{child-count}(B) = 3$) and the rest 2 intervals ($[3,5]$) to its right child C (as $\text{child-count}(C) = 2$). Then node B and C perform the similar interval assignments as A and finally the indexing intervals are assigned to all the sensors in the network.

4.3 Routing in the Interval Tree

After the interval distribution to all the sensors, we can start anywhere in the network and route to the query interval by traversing the tree. Starting from the query node, the idea is to forward the query upward the tree until finding an overlapping between the query interval and the children interval of some node, and then forward the query to the overlapped child downward the tree until we reach the sensor that indexes the interval. For instance, if node D wants to query interval $[3,4]$ which is maintained at node F . First D sends the query to its parent B as D is the leaf node and the query interval ($[3,4]$) does not overlap the interval maintained at D ($[0,1]$). Then B forwards the query to its parent A , as neither the Local Interval of B (i.e. $[2,3]$) nor any Children Interval of B (i.e. $[1,2]$) overlaps the query interval. A forwards the query to C as the Children Interval C (i.e. $[3,5]$) overlaps the query interval and then C forwards the query to F which can reply the query with the query results.

Lemma1: The communication distance to find an interval in the Interval Tree is at most $2L$, where L is the height of the tree.

4.4 Resilient Data Transmission

In this section, we present our mechanism to deal with link failures between sensors in the network. Recall that in the tree construction phase (Section 4.1), each node has

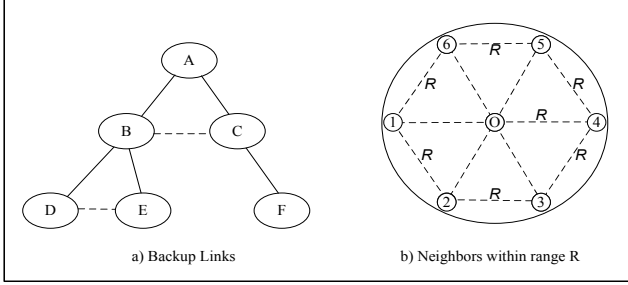


Figure 2. The *Backup Links* in the *Interval Tree*

several children and a parent in the tree as its neighbors. In reality, the node may have much more direct neighbors. This is because the edges in the *Interval Tree* may not be able to represent all the neighbor sensors that can communicate with each other directly. Take Figure 2 (a) as an example, node B and C and node D and E can reach each other in one hop as their distances are within the communication range R of the sensor. Figure 2 (b) shows that if a sensor (say O) has more than 6 direct children (as node 1, 2, 3, 4, 5, 6 in the figure, whose distance to O is less than R), there must exist a pair of children that can communicate with each other directly. In a dense sensor networks, we believe there are many such direct links that connect two nearby sensors but have not been represented in the *Interval Tree*. We call these direct links as *backup links* and take advantage of them to address link failures during routing. When some node or some communication link fails in the network (e.g. node A fails in Figure 2 (a)), its children detect the failure and utilize the *backup links* to route the message to the other part of the network (e.g. If node B wants to send a query to node F , it uses the *backup link* $B - C$ to route to F instead of routing to node A as node A is unavailable). Therefore, we can guarantee the reachability between two sensors within the sensor network if either the original link or the *backup links* along the path between these two sensors are functioning, which greatly improve the Resiliency of the data transmission.

4.5 Reliable Data Storage

We propose two reliable data sharing solutions, the *XOR_BACKUP* and *IDA_BACKUP* algorithms, to provide reliable storage of the sensor data in sensor networks. *XOR_BACKUP* utilizes a cheap and efficient logical XOR (i.e. exclusive or) operation to meet the critically limited computational capacity of sensor devices.¹ The idea is to perform a bitwise XOR operation in the *Interval Tree* from bottom to top and level-by-level. For each node in the *Interval Tree*, it XORs bit-by-bit all the backup

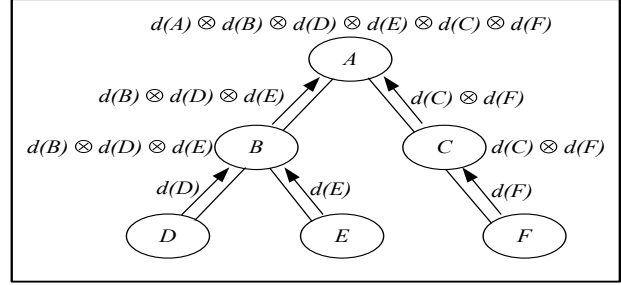


Figure 3. *XOR_BACKUP* in the *Interval Tree*

data sent from its direct children along with the local interval data it keeps, stores the result locally as backup information and sends the result to its parent. Take Figure 3 as an example, node D , E , F send their data to their respective parents as they are the leaf nodes of the *Interval Tree*. Node B , when receiving the data from node D and E , XORs the received data ($d(D)$, $d(E)$) and its own data $d(B)$. The result of XOR computation is then stored locally at node B and sends to B 's parent A . The *Root* A , which receives data from B and C , compute the XOR result of all the sensor data in the network. As A is the *Root*, it randomly chooses a direct child to store the backup information it keeps (i.e. $d(A) \otimes d(B) \otimes d(C) \otimes d(D) \otimes d(E) \otimes d(F)$).

With the backup information constructed in the *Interval Tree*, we can easily recover the missing data from node failures. For example, if node D fails in Figure 3, we can recover from the information at B and E ($d(D) = (d(B) \otimes d(D) \otimes d(E)) \otimes d(B) \otimes d(E)$). If an internal node B fails, we can recover the information stored at B by considering the information at node A , C , D and E (i.e. $d(B) = (d(A) \otimes d(B) \otimes d(C) \otimes d(D) \otimes d(E) \otimes d(F)) \otimes d(A) \otimes (d(C) \otimes d(F)) \otimes d(D) \otimes d(E)$).

After the recovery of the missing data, the neighbor node of the failed sensor stores the recovered data and takes care of the interval that used to be indexed by the failed node and the *XOR_BACKUP* is then performed again for the new topology to address possible failures in the future. For example, in Figure 3, if node D fails, its data will be recovered at its neighbor B and stored there. Node B will also index the interval that used to be indexed by node D and node B and A will re-compute the backup information since the data stored at B has changed.

The advantage of the *XOR_BACKUP* algorithm is that the computation of the backup information is very cheap and fast (as we just need to perform bit *xor* operation bit-by-bit on the data received from different sensors). *XOR_BACKUP* can also recover from multiple failures if the failed nodes are far from each other (i.e. they are neither siblings nor parent and child). However, it cannot recover current failures of "nearby" nodes (i.e. for any node S_i in the network, *XOR_BACKUP* cannot recover the

¹XOR is a logical bit operation on two one-bit operands that results in 1 if and only if exactly one of the operands has a value of 1.

missing data if two of its direct children fail, or one child and S_i fail). This is because the *XOR_BACKUP* algorithm segments the network into several groups with each group having one parent and all its direct children (i.e. each group is composed of "nearby" nodes) and the redundancy that *XOR_BACKUP* provides for each group is only capable of recovering from at most one failure in each group.

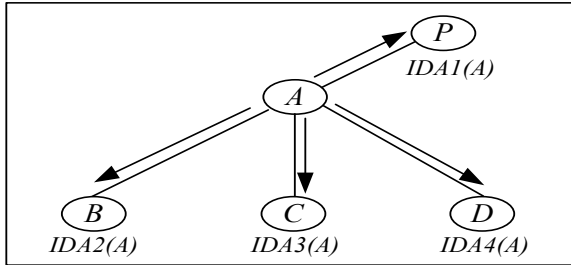


Figure 4. IDA_BACKUP in the Interval Tree

In applications where the failures occur frequently, *XOR_BACKUP* cannot recover all the missing data due to failures, we employ a more robust fault tolerance technique - the Rabin's Information Dispersal Algorithm (IDA) [14] to provide more reliability for the sensor data. The IDA algorithm is able to distribute a data of size L into N even-sized distributed shares (of size L/M each), in such a way that any subset of M of these N shares can recover the original data (we call this (M, N) -IDA code). The IDA code has two advantages that make it suitable for reliable data storage in sensor networks: (a) It is able to recover from multiple failures (i.e. it can recover from up to $N - M$ share failures, as we can reconstruct the original data from any M shares.) (b) The storage overhead is efficient compared with other techniques. To recover data of size L from $N - M$ failures, we only need to distribute LN/M redundant information in the system.

Now we present our *IDA_BACKUP* algorithm that utilizes IDA code to provide more reliability for the data storage in sensor networks. In the *IDA_BACKUP* algorithm, each node S computes the IDA codes of the data it keeps, and then distributes the IDA codes to its parent and all its direct children. In the IDA computation, the parameter N is chosen as the total number of the direct children along with the parent of S , and M is chosen as $M = N - f$ where f is the maximum number of simultaneous failures that *IDA_BACKUP* can recover. Take Figure 4 as an example, node A has 3 children B , C , D and a parent P . In order to provide more reliability for the data stored at A , we compute and distribute $N = 4$ shares of the data at A (i.e. $IDA1(A)$, $IDA2(A)$, $IDA3(A)$ and $IDA4(A)$) to all its children and its parent. Suppose we use $(2,4)$ -IDA code, if node A fails and any two of its children (say, B and D) also fail, we can still recover the data at A by using the redundancy data stored at the other two working nodes (say, C and P).

4.6 Load Balancing

In Section 4.2, we segment the data domain into n equal-sized intervals and assign each interval to an individual sensor. In this case, all the sensors will share the sensor data evenly if the generated sensor data distributes evenly over the data domain. In real world applications, however, this is not a reasonable assumption as the sensor data is dependent on the environmental changes. As a result, some intervals are more popular than others (i.e. they have more data records) and the sensors keeping these "popular" intervals require more storage overhead than other sensors, thus introduces large variation on the energy expenses at different sensors. Our objective is to segment the data domain into several (varying-sized) intervals and assign these intervals to sensors adaptively according to the data distribution in the sensor network in such a way that different sensors index and store sensor data of similar sizes.

In order to balance the data size kept at different sensors, we firstly assume that the generated sensor data fall in all the intervals with the same probability and then readjust the interval length kept at different nodes in order to accommodate the biased distribution of the sensor data. Our load balancing algorithm works by segmenting the "popular" intervals (i.e. those intervals that include more sensor data records than others) into several short intervals and merge multiple "non-popular" intervals into one long interval. The idea is to let the sensor communicate with its neighbors (i.e. children or parent) the size of the data it indexes and stores periodically (e.g. every day or every week), and if the difference between the data sizes of two nearby sensors (say, S_1 , S_2) surpasses a threshold τ , we will bipartition the "popular" interval I_1 (which kept at S_1) into two halves I'_1 and \bar{I}'_1 . The first half interval I'_1 is still kept at sensor S_1 while the other half \bar{I}'_1 along with the data it indexes is pushed to the "non-popular" sensor S_2 and maintained there. Please notice that this interval adjustment operation only requires an update on the interval information of the parent and child and doesn't affect any other node of the network, which introduces very small communication overhead.

5 Indexing Multi-dimensional Data

In Section 4, the data generated at each sensor is a single value and the query is a one dimensional interval. In real world applications, the sensors may be able to monitor multiple quantities of the environment at the same time (e.g. GPS locators can record the latitude and longitude of moving objects, environment sensors are able to measure the temperature, pressure and lighting for some area). The queries on these sensor data are therefore a multi-dimensional range query in a multi-dimensional data domain ($\bar{D} = D^t$). The problem here is to segment the multi-dimensional data domain \bar{D} into n sub-regions and assign

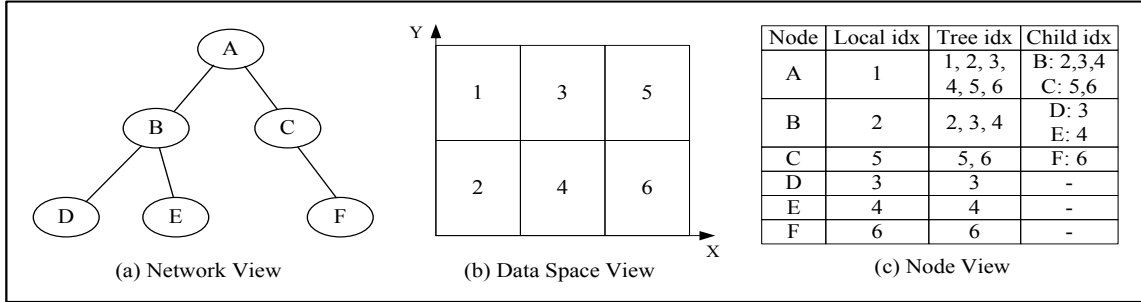


Figure 5. The Multi-dimensional *Interval Tree*

these sub-regions into different sensors, in such a way that we can answer multi-dimensional range queries effectively and efficiently.

Similar to Section 4.2, we initially assign all the n sensors n non-overlapping and even-sized sub-regions, and then adjust the sub-regions kept at different sensors dynamically in an online fashion according to the distribution of the newly generated sensor data. The initial assignment starts by segmenting the data domain D_t into n non-overlapping and even-sized sub-regions (e.g. In Figure 5, a two dimensional $X - Y$ data space is segmented into 6 even-sized sub-regions with region number from 1 to 6). These sub-regions are assigned node-by-node from the *Root* downside towards the leaf nodes. Each node \bar{S} , once receives the sub-regions assigned from its parent, assigns these sub-regions to its direct children proportionally to the cardinalities of the sub-trees rooted from them. The information kept at each node \bar{S} in the *Multi-dimensional Interval Tree* is as follows: (a) a Local Index, indicating the sub-region that node \bar{S} keeps; (b) a Tree Index, which keeps the sub-regions that the sub-tree rooted from \bar{S} will keep; (c) a Child Index, keeping the Tree Indices for each direct children of \bar{S} . Take Figure 5 as an example, initially *Root A* takes the whole data space as Tree Index and keeps sub-region 1 as its Local Index and assigns sub-region 2,3,4 to child *B* (as there are 3 nodes in the sub-tree rooted from *B*) and 5,6 to child *C* respectively. Node *B* and *C*, once receives the Tree Index (i.e. the Child Index of *A*) assigned to them, take one sub-region as their Local Index and then assign the others to their children accordingly. Once all the sub-regions have been assigned to all the sensors, we can perform routing in the network (to answer various queries for multi-dimensional data) in a similar way as described in Section 4.3.

6 Experimental Evaluation

In this section, we present an extensive performance evaluation of the *Interval Tree* algorithm and the *GEM* technique using real world data sets. Our goal is to demonstrate that our *Interval Tree* algorithm is an effective and efficient data-centric storage system for sensor networks and it is capable of recovering the missing data due to failures. We

have implemented a trace driven simulator in GNU C++ (version 3.4.5). The simulation was performed on a PC with an Intel Pentium 4 (2.8 GHz) CPU and 1 GB of memory.

6.1 Data sets

We use two data sets for our experiments:

Intel: This real world data set is a trace of sensor readings collected from 54 sensors deployed in the Intel Berkeley Research lab between February 28th and April 5th, 2004.² The sensors collected timestamped humidity, temperature, light and voltage values in 31 second intervals. We construct the network topology according to the location of the sensors and randomly choose a sensor as the root of the *Interval Tree*.

Random: There are a total of 100 sensors distributed evenly in a 100×100 grid. We adjust the network density and topology by changing the communication range of a sensor (i.e. the longest distance for two sensors to communicate directly). To simulate the real sensor measurements at different spots, we generate both random and biased distribution of the data records at each sensor.

6.2 Comparison Systems

We evaluate the performance of our *Interval Tree* algorithm and two other strategies for query processing in sensor networks:

GEM: This is the graph embedded routing and data storage mechanism proposed in [12]. The *GEM* algorithm constructs a labeled graph that is embedded into the network topology in a distributed fashion. Compared with *Interval Tree*, *GEM* only stores data at leaf nodes of the tree which wastes storage capacity of the internal nodes. In addition, *GEM* does not provide reliable data storage against failures and dynamic load balancing on the sensor data.

Localized: This is a simple data storage scheme that stores data where they are generated without indexing them. It introduces no communication overhead for data storage

²<http://berkeley.intel-research.net/labdata/>

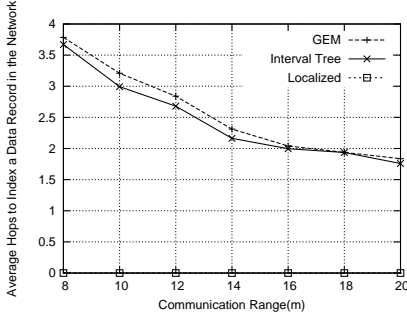


Figure 6. Average data indexing overhead (Intel data)

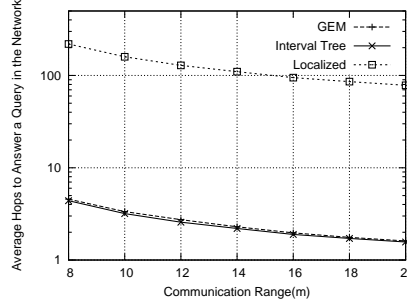


Figure 7. Average *Equality Query* overhead (Intel data)

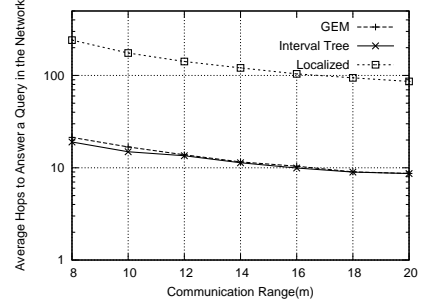


Figure 8. Average *Range Query* overhead (Intel data)

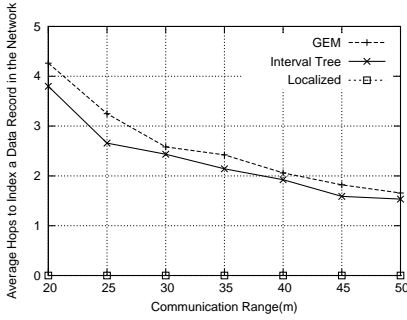


Figure 9. Average data indexing overhead (Random data)

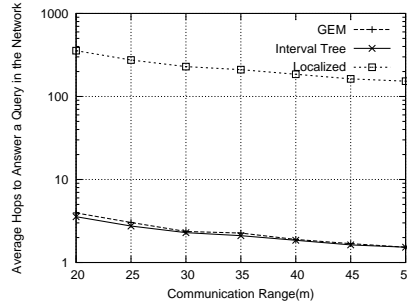


Figure 10. Average *Equality Query* overhead (Random data)

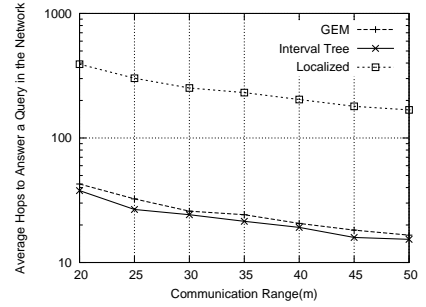


Figure 11. Average *Range Query* overhead (Random data)

but requires accessing all the sensors in the network when processing a query.

6.3 Overhead of data Indexing in the network

In data-centric storage systems, whenever a new data record is generated at a sensor, it will be transferred somewhere in the network for efficient query processing. We call the overhead due to this type of data transmission as data indexing overhead. With the same network settings, we compare the communication overhead of data indexing for *Interval Tree*, *GEM* and the *Localized* approach where each data record is kept locally at the sensor that generates it. We vary the communication range of sensors (e.g. $8m \sim 20m$ for Intel Data and $20m \sim 50m$ for Random Data) in order to evaluate the data indexing overhead under various network topologies. As shown in Figure 6 and 9, *Interval Tree* always introduces less data indexing overhead than *GEM*. This is because the *Interval Tree* uses the internal nodes in addition to leaf nodes for data indexing and thus decreases the average distance to index a data record. The *Localized* approach always has no indexing overhead in terms of communication as the data records are maintained locally at the generation sensors without any transmission.

6.4 Overhead of query processing in the network

Given the domain of the sensor data, we generate 10,000 random *Equality Queries* and *Range Queries* on the data stored in the sensor network. Each query is issued from a random sensor in the network and we measure the average communication overhead for each query. Figure 7, 8, 10 and 11 show that *Range Queries* usually require more query overhead than *Equality Queries* as more queries results in *Range Queries* are transmitted to the query node. The *Localized* approach requires more than an order of magnitude additional query overhead than *GEM* and *Interval Tree*. This is because *Localized* requires broadcasting the query to all the sensors in order to answer a query while the data-centric approaches only access the node indexing and storing the query value. In addition to that, the *Interval Tree* algorithm outperforms the *GEM* technique by having less average communication overhead to process a query. This is also due to the employment of the internal tree as indexing node in the *Interval Tree*.

6.5 Failure Recovery

We evaluate and compare the performance of two data recovery techniques: *XOR_BACKUP* and *IDA_BACKUP*. As shown in Figure 12 and 15, with increasing failure rate

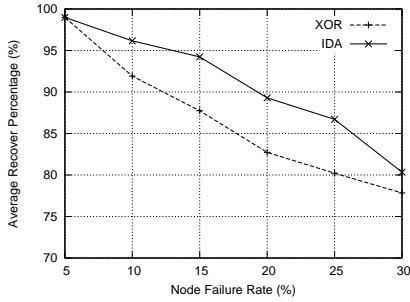


Figure 12. Varying failure rates (Intel Data)

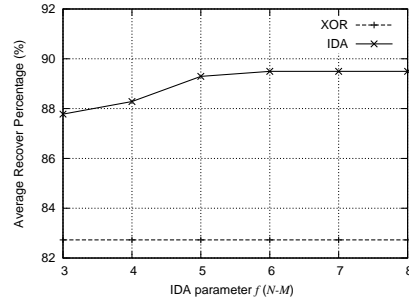


Figure 13. Varying the parameter f (Intel Data)

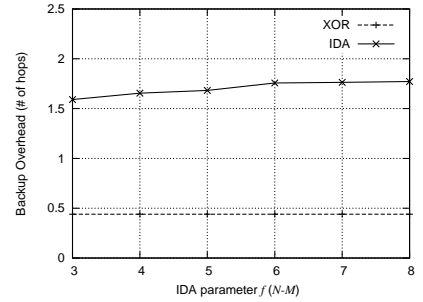


Figure 14. Storage overhead of XOR and IDA (Intel Data)

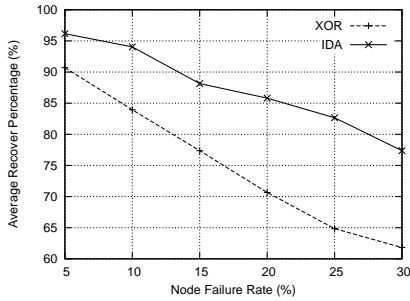


Figure 15. Varying failure rates (Random Data)

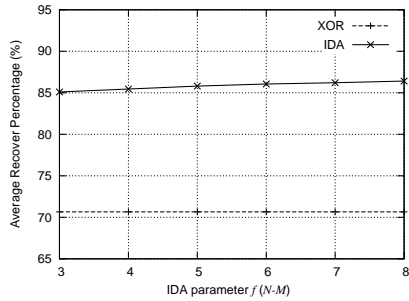


Figure 16. Varying the parameter f (Random Data)

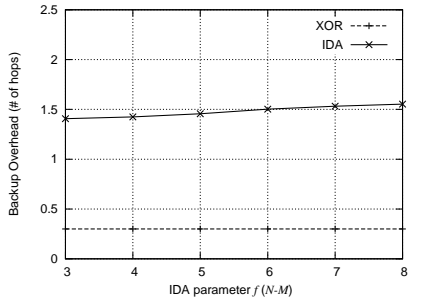


Figure 17. Storage overhead of XOR and IDA (Random Data)

of the sensor nodes, both techniques are able to recover less data. *IDA_BACKUP* is able to recover more failure sensor data than *XOR_BACKUP*. We also vary the parameter f for *IDA_BACKUP* (recall that $f = M - N$, is the maximum number of failures that can be recover within each group). Figure 13, 14, 16 and 17 show that with larger value of f , *IDA_BACKUP* is able to recover more failures but introduces more communication overhead (i.e. the average number of hops to store the redundancy information).

6.6 Load balancing

In order to prolong the lifetime of the sensor networks, we need to balance the size of the data kept at different sensors so that the energy spent at different sensors will be similar. In this experiment, we evaluate the performance of our load balancing algorithm for the *Interval Tree*. As shown in Figure 18 and 19, even without load balancing, *Interval Tree* outperforms *GEM* in terms of load balancing (less variance in the data size stored at each sensor), as *GEM* does not use internal nodes to index data. With the load balancing mechanism, the *Interval Tree* is able to balance the data size maintained at different sensors much better. Figure 18, 19 and 20 show that, with increasing update threshold, the load balancing algorithm show less balancing power but also introduces less overhead (in terms of communication).

This is because less and grosser balancing operations are performed when the update threshold becomes larger.

7 Conclusions

In this paper we introduced and formalized the reliable and load balancing data-centric storage problem for sensor networks. We proposed a data-centric storage and query routing algorithm, *Interval Tree*, that provides an efficient and effective solution to this problem. The routing mechanism in *Interval Tree* is constructed upon the neighborhood information of each sensor and is completely independent on any geographical information. The distribution of intervals over sensors constructs a data-centric storage system in the network which provides efficient routing and processing of queries. In addition, *Backup links* and node backup mechanisms guarantee the resilient transmission and reliable storage of the sensor data. Finally *Interval Tree* also provides an adaptive load balancing mechanism that is able to adjust the interval length indexed at different sensors according to the distribution of the new generated sensor data. For future work, we plan to explore a more efficient interval distribution mechanism so that the routing distance between the sensor node generating the data and the index node indexing and storing the data is minimized.

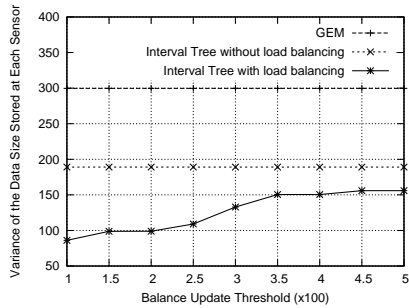


Figure 18. Variance of the data sizes at sensors (Intel Data)

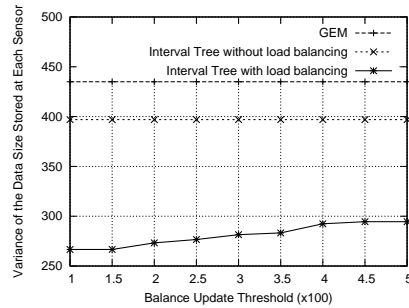


Figure 19. Variance of the data sizes at sensors (Random Data)

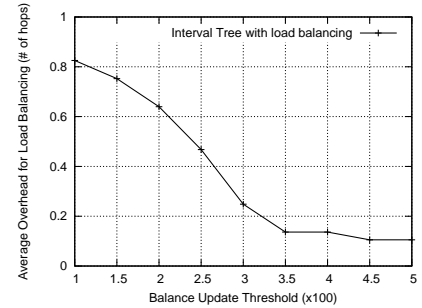


Figure 20. Average communication overhead for load balancing (Intel Data)

Acknowledgement

This work was supported by NSF (IIS 0330481, IIS-0534781).

References

- [1] J. C. Abhishek Ghose, Jens Grossklags. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proceedings of Mobile Data Management*, pages 45–62, 2003.
- [2] M. Aly, K. Pruhs, and P. K. Chrysanthis. Kddcs: a load-balanced in-network data-centric storage scheme for sensor networks. In *Proceedings of CIKM*, pages 317–326, 2006.
- [3] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of Mobile Data Management*, pages 3–14, 2001.
- [4] H. Dai and R. Han. A node-centric load balancing algorithm for wireless sensor networks. In *IEEE Global Telecommunications Conference*, pages 548–552, 2003.
- [5] P. Hsiao, A. Hwang, H. Kung, and D. Vlah. Load-balancing routing for wireless access networks. In *INFOCOM*, pages 986–995, 2001.
- [6] H. Kakiuchi. Dynamic load balancing in sensor networks. Technical report, Stanford University, 2004.
- [7] B. Karp and H. T. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In *MobiCom*, pages 243–254, 2000.
- [8] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of Sensys*, pages 63–75, 2003.
- [9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.
- [10] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *ACM SIGMOD*, pages 491–502, 2003.
- [11] A. Mitra, A. Banerjee, W. Najjar, D. Zeinalipour-Yazti, D. Gunopulos, and V. Kalogeraki. Rise co-s : High performance sensor storage and co-processing architecture. In *SECON'05*, 2005.
- [12] J. Newsome and D. Song. Gem: graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Sensys*, pages 76–88, 2003.
- [13] S. Park, R. Vedantham, R. Sivakumar, and I. Akyildiz. A scalable approach for reliable downstream data delivery in wireless sensor networks. In *Proceedings of MobiHoc*, pages 78–89, 2004.
- [14] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of ACM*, 36(2):335–348, 1989.
- [15] S. Ratnasamy, B. Karp, S. Shenker, D. Estrin, R. Govindan, L. Yin, and F. Yu. Data-centric storage in sensor networks with ght, a geographic hash table. *Mobile Network Applications*, 8(4):427–442, 2003.
- [16] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceeding of Middleware*, pages 329–350, 2001.
- [17] K. Seada, A. Helmy, and R. Govindan. On the effect of localization errors on geographic face routing in sensor networks. In *IPSN*, pages 71–80, 2004.
- [18] K. Seada, M. Zuniga, A. Helmy, and B. Krishnamachari. Energy-efficient forwarding strategies for geographic routing in lossy wireless sensor networks. In *Sensys*, pages 108–121, 2004.
- [19] M. Sharaf, J. Beaver, A. Labrinidis, and P. Chrysanthis. Tina: A scheme for temporal coherency-aware in-network aggregation. In *MobiDE*, 2003.
- [20] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensor networks. *SIGCOMM Computer Communication Review*, 33(1):137–142, 2003.
- [21] F. Simjee, D. Sharma, and P. H. Chou. Everlast: long-life, supercapacitor-operated wireless sensor node. In *SenSys*, page 315, 2005.
- [22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.
- [23] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.
- [24] Y. Yao and J. Gehrke. Query processing in sensor networks. In *In Proc. of the CIDR*, 2003.
- [25] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report, UC Berkeley, Apr 2001.