Seventh IEEE International Symposium on Network Computing and Applications

Sorting Based Data Centric Storage

Fenghui Zhang, Anxiao(Andrew) Jiang, and Jianer Chen
Dept. of Computer Science, Texas A&M Univ. College Station, TX 77843. {fhzhang, ajiang, chen}@cs.tamu.edu.

***Abstract*— Data-centric storage [6], which supports efficient in-network data query and processing, is an important concept for sensor networks. Previous approaches mostly use hash functions to store data, where data with the same key value are stored in sensors at or near the same geographic location.**

We propose a new data-centric storage method based on sorting. Our method is robust for different network models and works for unlocalized homogeneous sensor networks, i.e., it requires no location information. The idea is to sort the data in the network based on their key values, so that queries – including range queries – can be easily answered. The sorting method balances the storage load well. We present a sorting algorithm that is both decentralized and efficient.

I. Introduction

Wireless sensor networks are widely deployed nowadays to collect data, aggregate data and answer queries. For sensor networks, where and how to store data is an important issue.

The *in-network storage* is one of the solutions for data storage, in which data are stored on the sensors themselves. It removes the dependency on servers, and balances power consumption better. It also supports in-network processing well, which is important for real-time applications. Like a database, the data in a sensor network are labelled by their key value. To answer queries efficiently, the data with the same key value should be aggregated and stored in the same place, and that place should be easily accessible by any sensor querying the data. That is the basic principle of *data-centric storage* [6], which has been a well accepted concept in sensor networks.

There have been a number of data-centric storage schemes proposed in recent years [2], [5]. Their main technique is the *geographic hash table*, where the location to store the data with the same key value is determined by a hash function. Any sensor querying the data can use the same hash function to find the location of the data.

Although the hash-based method is an elegant solution for data-centric storage, two difficulties still remain: *storage load balance* and *the support for range queries*. When the data with some key values are more abundant than the data with other key values, the sensors corresponding to the first group of data need to store more data than the second group. Also, since the hash function builds a simple mapping between keys and locations, when there is a hole in the sensor network, the sensors around the hole often need to store much more data than others. And in fact, our simulations show that even the random deployment of sensors alone can lead to substantial load balancing problems. On the other hand, since the hash function maps similarly key values independently to different locations, range query (a sensor queries the data whose key values are in a range) becomes expensive. With the hash-based method, for every key value in the range, a separate query message needs to be sent, even if the data of some key values do not exist. Therefore, the communication cost can be high.

In this paper, we propose a new data-centric storage method: *sorting-based storage*. The idea is to sort the data in the network based on their key values. A primary path in the network is used to provide a linear order of the data, and all the edges are used to make both the sorting process and the query process very efficient. The sorting process naturally balances the storage load for sensors very well, regardless of the distribution of data or the shape of the network. And since adjacent data are stored sequentially in nearby sensors, range queries can be easily answered. We present a sorting algorithm that is both decentralized and very efficient. It requires no location information, and is robust to different network models.

Due to the page limit, we have skipped many details. Interested readers are referred to [1] for a complete version.

II. Basic Concepts and Terms

Consider a network G with n nodes and m data objects (each with an integer key). The network contains a path P of length N that goes through every node at least once. The NETWORK SORTING problem asks to store the data so that the order of their key values is consistent with the order of the corresponding nodes in the path.

In the following we define two terms used for evaluating the complexity of an asynchronous algorithm.

Definition 1: Given a distributed network G with n nodes, during an asynchronous computational process in the network each node takes actions. We call it a *round* when every node has taken exactly one action or given up the chance to take an action. We call each action a *step*.

We will use the term *load factor* to measure the load balance of a data storage scheme, defined as follows.

Definition 2: Given a storage scheme where the average load is d data objects per node, the maximum load is d_{max} data objects in a node and the minimum load is d_{min} data objects in a node, the *load factor* of the scheme is then $\max\{\frac{d_{max}}{d}, \frac{d}{d_{min}}\}$.

In order to construct the path P when the node positions are not know, we will need some concepts related to planar graphs, which are defined below.

Definition 3: Given a planar graph G_p embedded in the plane, a face F is *adjacent* to another face F' if F and F' share an edge or a vertex. Similarly we say that a face F is *adjacent* to a path P if F and P share an edge or a vertex. The *distance* from a face F to a path P is the minimum number

978-0-7695-3192-2/08 $25.00 © 2008 IEEE
DOI 10.1109/NCA.2008.16

of faces we need to walk through in order to walk from F to a face adjacent to P (by walking through adjacent faces, not counting F itself). The *distance* between an edge e and a path P is the smallest value among the distances from the faces adjacent to e to the path P. The distance between two paths P, P' is the smallest value among the distances from the edges in P to the path P'.

III. Network Sorting and Balanced Data Storage

To solve the NETWORK SORTING problem, we first describe a distributed algorithm for the case where P is a Hamiltonian path (i.e., each node appears in P exactly once, $n = N$) and $n = m$. We will extend the algorithm to solve the general general NETWORK SORTING problem.

During the sorting process, every node remembers the keys of its own data and its neighbors' data. The basic step in the sorting algorithm is a *local sorting operation*:

- A node sorts the data within 1-hop based on the positions of the nodes in the path. After the sorting, each node whose data has changed informs its neighbors of the key of its new data.

The key of the algorithm is to carry out the local sorting operations asynchronously and efficiently. Our algorithm utilizes a modified scheduling scheme from the d-scheduling algorithm described in [4]. Suppose a node u's order in the path P is i, we assign the node u a priority i. In the sorting algorithm, the smaller a node's priority is, the higher a priority the node has for carrying out a local sorting operation.

The general process of sorting with scheduling is as follows:

- Each node in whose neighborhood a local sorting is necessary requests a local lock to its two hop neighborhood. Let u be a node whose priority is the highest among the nodes within two hops that have sent lock request. The node u will then lock its 2-hop neighborhood and perform a local sorting operation. After the operation, node u increases its priority by n and release the lock.

When a node u is sorting data, its neighbors can neither sort data nor have their data sorted by any node other than u. The 2-hop neighbors of u cannot perform sorting, but their data may be sorted by other nodes.

During the sorting process, the priorities of all the nodes are in total order at any moment. The dependent graph obtained from the network G by putting a directed edge from a node to each neighbor with a higher priority is always a DAG (directed acyclic graph). Therefore there will be no deadlock and the termination is guaranteed. Here we call the actual local sorting operation performed by a node in the network a *step*.

The following theorem summarizes an general bound for the number of rounds needed to sort all the data objects in the network. Due the space limit, we skip the proofs for all lemmas and theorems in this paper. Interested readers are referred to [1] for details.

Theorem 1: The sorting algorithm described above solves the NETWORK SORTING problem for $N = n = m$ after at most N rounds.

Each round consists of at most N steps, the following corollary is then straightforward.

Corollary 1: The sorting algorithm described above solves the NETWORK SORTING problem when $N = n = m$ with at most $O(N^2)$ steps.

Now let us consider the case where P is not a Hamiltonian path (i.e., $N > n$) and $m = N$. By letting each node act as multiple virtual nodes, one for each occurrence on the path P and letting the distance among all virtual nodes on the same actual node be 0, the previous algorithm can be applied to solve the sorting problem. The algorithm will again terminate in $O(N)$ rounds and $O(N^2)$ steps.

In the following we consider the general case when there is no constraint for n, N and m. Before we sort the data, each node learns the values of n, N and m. Then every node knows that it node should have $\lceil m/n \rceil$ or $\lfloor m/n \rfloor$ data objects in total after sorting. During the sorting process, every node u keeps track of the number of data objects stored by each of its virtual neighbors. When a node u performs the local sorting operation, u collects all the keys in its neighborhood, sorts them, and re-distribute the data objects so that every neighbor and itself store the data as evenly as possible.

Theorem 2: The sorting algorithm described above solves the NETWORK SORTING problem in $O(Nm)$ rounds and $O(N^2m)$ steps.

The upper bounds presented so far for the sorting performance are for general networks. In the following, we analyze the sorting performance for one- and two-dimensional arrays. We summarize the results for the previously discussed simplified model, where the path P is a Hamiltonian path, and the number of data objects equals the number of nodes. That is, $n = N = m$. These results provide evidence that the actual sorting performance in sensor networks can be much better than the upper bounds presented earlier, because sensor networks often resemble two-dimensional mesh networks.

A. Sorting in linear array

When the underlying network is a linear array of n nodes, our algorithm has the same performance as the odd-even transposition sort algorithm described in [3].

Theorem 3: The NETWORKSORTING algorithm solves the network sorting problem on a linear array of size n in $O(n)$ rounds.

B. Sorting in $\sqrt{n} \times \sqrt{n}$ grid

Now suppose our network is a $\sqrt{n} \times \sqrt{n}$ grid and the path P is a snake-like path [3]. We list a few lemmas first.

Lemma 1: Suppose the data to be sorted are only 0s and 1s. Suppose in a $n_1 \times n_2$ grid ($n_1 \leq n_2$), all the data objects in the first i ($1 \leq i < n_1$) rows are sorted according to the snake-like path P, and so are all the data objects in the last $n_1 - i$ rows. The NETWORKSORTING algorithm will sort all the data objects in the grid in $2n_2$ rounds.

Lemma 2: The NETWORKSORTING algorithm solves the network sorting problem on a $\sqrt{n} \times \sqrt{n}$ grid with data being 0s and 1s in $O(\sqrt{n} \log n)$ rounds.

By the 0-1 sorting lemma [3], we get have the following theorem.

Theorem 4: The NETWORKSORTING algorithm solves the NETWORK SORTING problem on a $\sqrt{n} \times \sqrt{n}$ grid in $O(\sqrt{n}\log n)$ rounds and $O(n \log n)$ steps.

The above upper bound for the NETWORKSORTING algorithm matches the performance achieved by the Shear Sort algorithm [3], which requires global synchronization. Since the NETWORKSORTING algorithm is asynchronous, it is more appropriate for sensor networks.

IV. IMPLEMENTATION

In this section, we give a sketch of our sorting-based balanced data storage scheme. For details, please refer to the complete version of this paper [1]. The storage scheme stores data in a sorted and load-balanced way. The former property ensures that queries, including range queries, can be efficiently answered. In addition to presenting the construction of the path P, we also discuss other aspects of the data-centric storage scheme.

A. Constructing the path P

The actual complexity of the sorting algorithm relies on the shape and length of the path P. A short path with many shortcuts is desirable. Finding the shortest path that traverses all nodes in a given graph is NP-hard. On the other hand, the length of the path is at least n, and the traversing of any spanning tree of the network will give us a path of length roughly $2n$. This gives us a simple ratio-2 approximation for the path.

In this subsection, we present a practical algorithm that constructs the path P, which is typically much shorter than $2n$ for unlocalized wireless sensor networks.

The algorithm for finding such a path consists of three major steps: (1) planarize the network; (2) construct a snake-like backbone path in the planarized network; (3) extend the path to include all nodes in the original network.

There have been several papers on how to obtain a planarized network efficiently for unlocalized wireless sensor networks [7], [8]. We skip the first step here. In the following we will discuss the second and the third steps.

1) Construct the backbone path: Assume that the network is already planarized and we have a topological embedding of the planar network, denoted by G_{planar}. To simplify the discussion, we assume that the outer face of the planar graph G_{planar} is a simple cycle.

Let f be the outer face of the network G_{planar}. We split f into four path segments P_1, P_2, P_3, P_4 of roughly equal lengths. Next we stretch the outer face into a square such that P_1, P_2, P_3, P_4 are on the north, east, south and west side of the square, respectively. In the following discussion, the outer face is not to be considered in super paths or the measurement of distances.

The medial axis of two paths is defined as a path that is of roughly the same distance from those two paths. The following

Algorithm 1 Medial-Axis

Input: G_{planar}, P_1, P_3

Output: P': a path formed by nodes whose distances to P_1, P_3 are roughly equal.

1: $G' \leftarrow$ the subgraph induced by faces and edges (excluding the outer face and its edges) whose distances to P_1 and P_3 are equal.
2: Let u, v be two nodes in G' that are on the east and west boundaries of G_{planar}, respectively.
3: **return** a path P' from u to v in G'

Medial-Axis algorithm is used repeatedly in constructing the backbone path.

The path P' separates the network G_{planar} into two subgraphs. We recursively partition the network and find medial axis paths. In the end, we get a set of "horizontal paths" in the network G_{planar} that do not cross each other.

The following is the algorithm for constructing all the horizontal paths in the network.

Algorithm 2 Horizontal-paths

Input: G_{planar}, P_1, P_3

Output: Horizontal paths containing all the nodes in G_{planar}.

1: **if** $P_1 = P_3$, **return**
2: $P' \leftarrow$ Medial-Axis(G_{planar}, P_1, P_3)
3: Output P'
4: $G_1 \leftarrow$ the subgraph between P_1 and P'
5: Horizontal-paths(G_1, P_1, P')
6: $G_2 \leftarrow$ the subgraph between P' and P_3
7: Horizontal-paths(G_2, P', P_3)

Now we have obtained a set of horizontal paths that do not cross each other. They form a total order from top to bottom. A straightforward way to get the snake-like path is to take the first path from left to right, connect it to the right end of the second path (which is simple to do), and walk through the second path from right to left, then turn on the third path in a similar way, and so on.

2) Construct the path traversing all nodes: The path constructed so far may not include all the nodes of the original network G. We use the following simple heuristic to include all nodes. If a node has two neighbors that are consecutive in the path, it inserts itself into the middle of the neighbors. After that, if there are still a few nodes not in the path, use a tree to attach to the path, and use the traverse of the tree as part of the path.

B. Sorting data in the network

We assume that each node knows the total number m of data objects stored in the network, the number n of nodes in the network, and the length N of the path P. (These numbers can be easily learned with a simple information collection operation, especially with the help of the path P.) Then every node should store $\frac{m}{n}$ data objects. A node knows how many times it appears in the path, so it can decide how many data objects to assign to each of its occurrences in the path. The

sorting process is described as before. It generates a sorted and balanced storage result.

C. Data access and query

With the sorted data and the path P, querying data is simple. When a node u wants to query data objects with the key value k, it sends a query message that contains the value k and its own address on P. Every node relays the query to the neighbor whose data objects have a key closest to k as the next hop. The process continues until the query message reaches the destination node. The routing is guaranteed to succeed because the edges in P can always be used for routing if necessary. In practice, most of the time the edges not in P are *shortcuts* and make routing much more efficient. To send the data back to u, a similar routing protocol is used, except that node u's address on P is used for routing instead of the key k. And the routing is also guaranteed to succeed.

Ranged queries are answered in a similar manner, because data with adjacent key values are stored next to each other in the path P.

D. Data dynamics

When a data object of key k is inserted into the network, the initiator of the insertion will send the object as if it were a query message for k. Once the message reaches its destination (where it should be stored), the new data object will be stored there.

The total number of the data objects in the network will be calculated and broadcasted periodically. With the existence of the path P, this task is simple. Once a node discovers that its load is too heavy or too light, it first tries to solve the imbalance locally by exchanging data with its neighbors on the path P. If local operations fail to bring back load balance, it triggers a re-balance process. This process is similar to the sorting process, only cost less.

Data deletion is dealt in a very similar way as insertion.

V. Performance Evaluation

We conducted extensive simulations to test the performance of our sorting based data centric storage scheme. The results show that the performance is very stable for different network models and different degree of data loads. We briefly compare our results with GHT. Please refer to [1] for detailed results.

We randomly deploy $n = 1500$ nodes in the sensor field. The network models we explore include unit-disk graphs (UDG) and quasi-unit disk graphs (quasi-UDG). We also test the storage schemes with holes being present in the sensor field. The data objects are generated by nodes randomly with keys in the range $[0, n]$. The total number of data objects we examined ranged from $100n$ to $500n$. Table I shows the data load balance performance of our scheme comparing to that of GHT with average load 500.

We compare GHT to sorting based storage scheme. For the sorting-based schemes, we allow the storage load of neighboring nodes to differ by at most $\delta = 4\%$, and no nodes' storage load is allowed to be 1.5 time or more than the average load. We distribute $500n$ random data objects in the network. Our results shows that the average load factor of the sorting based scheme is always close to 1, while that of GHT can be very large (Fig. 1(a)).

	UDG		UDG(hole)		2-qUDG	
	max	σ	max	σ	max	σ
GHT	4271.18	577.56	4737.01	592.03	3740.59	571.05
Sorting	564.42	6.18	561.50	6.10	560.17	5.54

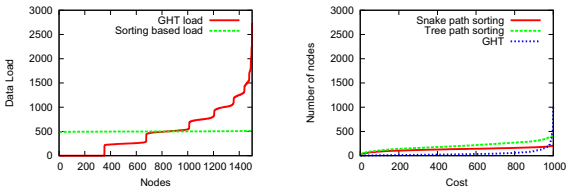
TABLE I
COMPARISON OF THE MAXIMUM AND THE STANDARD DEVIATION OF THE STORAGE LOAD WITH AVERAGE LOAD 500.

Scheme	operation	UDG	UDG(hole)	2-qUDG
GHT	*	291.14	372.47	59.45
Sorting	query	182.37	175.97	142.77
	insertion	67.62	66.14	62.20

TABLE II
COMMUNICATION COST FOR STORAGE AND QUERY: AVERAGE NUMBER OF MESSAGES PER DATA OBJECT PER OPERATION

We compare the average communication costs for the three schemes. The average cost of the sorting scheme is substantially lower than that of GHT (Table II).

Fig. 1 (b) shows the distribution of the distance of data travelled during the storage process.



(a) (b)

Fig. 1. (a) Typical data load distributions of the GHT scheme and the sorting(snake) scheme in UDG networks with average degree approximately 7; (b) The distribution of the distance of 1000 sample data objects travelled during the storage process.

References

[1] F. Zhang, A. Jiang and J. Chen, Sorting based data centric storage, *Technical Report, TR2008-5-1, Department of Computer Science, Texas A&M University*, (http://www.cs.tamu.edu/academics/tr/tamu-cs-tr-2008-5-1), May 2008.

[2] J. Newsome and D. Song, GEM: Graph EMbedding for routing and data centric storage in sensor networks without geographic information, *Proc. Sensys 2003*, pp. 76-88, 2003.

[3] F. T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes *San Mateo, CA: M. Kaufmann Publishers*, 1992.

[4] Y. Malka, S. Moran and S. Zaks, A lower bound on the period length of a distributed scheduler, *Algorithmica*, 10(5), pp. 383-398, 1993.

[5] S. Ratnasamy, L. Yin, F. Yu, D. Estrin, R. Govindan, B. Karp, S. Shenker, GHT: A geographic hash table for data-centric storage, *Proc. WSNA*, pp. 78-87, 2002.

[6] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin, Data-centric storage in sensornets, *ACM SIGCOMM HotNets*, 2002.

[7] Y. Wang, J. Gao, and J. S.B. Mitchell, Boundary recognition in sensor networks by topological methods, *Proc. MobiCom*, pp. 122-133, 2006.

[8] F. Zhang, A. Jiang and J. Chen, Robust planarization of unlocalized sensor networks, to appear in *Proc. INFOCOM*, Phoenix, Arizona, 2008.