EnviroStore: A Cooperative Storage System for Disconnected Operation in Sensor Networks

Liqian Luo, Chengdu Huang, Tarek Abdelzaher
Department of Computer Science
University of Illinois at Urbana-Champaign
{lluo2, chuang30, zaher}@cs.uiuc.edu

John Stankovic
Department of Computer Science
University of Virginia
stankovic@cs.virginia.edu

Abstract— **This paper presents a new cooperative storage system for sensor networks geared for disconnected operation (where sensor nodes do not have a connected path to a basestation). The goal of the system is to maximize its data storage capacity by appropriately distributing storage utilization and opportunistically offloading data to external devices when possible. The system is motivated by the observation that a large category of sensor network applications, such as environmental data logging, does not require real-time data access. Such networks generally operate in a disconnected mode. Rather than focusing on multihop routing to a basestation, an important concern becomes (i) to maximize the effective storage capacity of the disconnected sensor network such that it accommodates the most data, and (ii) to take the best advantage of data upload opportunities when they become available to relieve network storage. The storage system described in this paper achieves the above goals, leading to significant improvements in the amount of data collected compared to non-cooperative storage. It is implemented in nesC for TinyOS and evaluated in TOSSIM through various application scenarios.**

I. Introduction

Data collection has been addressed at length in sensor network literature. The assumption has traditionally been that a sensor network is connected to a basestation that collects the data. This assumption is suitable when near-real-time information availability is desirable. Tracking and event notification applications, for example, fall under this category. A significant number of applications, however, do not require real-time information [17][26][10]. For example, an environmental scientist interested in studying light variations on the forest floor due to canopy closure in the Spring might deploy a sensor net and collect the data only months later when the experiment is over[1]. This "fisherman's net" model of the application allows for significant architectural simplifications of the monitoring infrastructure. Most importantly, there is no longer a need to maintain a basestation in the field, which is very convenient. A user no longer has to worry about powering up the basestation in the wilderness (without power outlets), protecting it from harsh weather and animals, and enduring the risk of losing data because of a centralized point of failure. Indeed, all that is needed is in-network storage (already available on sensor nodes as flash memory) and a capability for opportunistic data upload. We call the above, a *disconnected* network model.

The disconnected model does not preclude sporadic contact with a basestation during network lifetime. For example, a user may choose to visit the field periodically for maintenance purposes (e.g., to remove dirt and debris that may occlude light sensor inputs over time). Such visits may be used for opportunistic data upload. The user could carry a data mule device [21] that collects data wirelessly from encountered nodes and dumps these data later to the basestation (e.g., a computer in the user's office).

A primary concern of the sensor network in this model becomes that of maximizing effective storage capacity (i.e., minimizing data loss due to flash memory overflow while the network is not connected). Observe that some nodes will record more data than others. This may be due to asymmetry in environmental inputs (e.g., acoustic nodes near sound sources will fill up before those in quiet areas), or due to data-dependent variations in compression ratio of algorithms such as run-length encoding. Data loss can be minimized by migrating data from nodes that are full to those that are not, as well as by exploiting upload opportunities when available.

In this paper, we present *EnviroStore*, a cooperative storage system for sensor network applications geared for disconnected operation. EnviroStore employs data redistribution schemes to optimize sharing of network storage. The amount of data that can be stored in the network is also affected by the power consumption of nodes. Naturally, nodes will be unable to record data after energy is depleted. EnviroStore takes into account the rate of energy consumption to avoid depletion-related data loss. Evaluation shows that in networks with a large input data imbalance, EnviroStore can delay the onset of data loss by nearly an order of magnitude.

EnviroStore reflects a change in paradigm for sensor network operation from communication-centric to storage-centric. Indeed, the growing size of low-power flash memory suggests that future nodes will have a much larger storage compared to their communication bandwidth. With the increase in storage capacity, new higher-bandwidth sensing modalities (such as multimedia) will undoubtedly be deployed that take advantage of the extra storage space. This will further exacerbate the communication bottleneck as low-power radio bandwidth does not grow at the same rate as low-power storage capacity.

[1] This is an actual study that has been performed in Trelease Woods near UIUC campus in Spring 2006

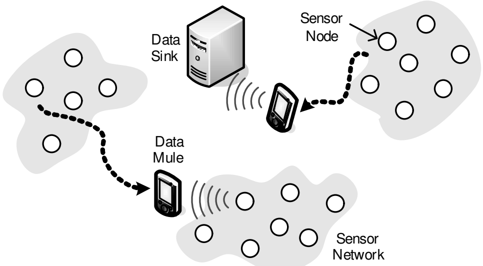


Fig. 1. System Model

Multihop communication will need to be minimized. Physical data collection via data mules will become more common, motivating data services such as those described in this paper.

A storage-centric paradigm for sensor networks differs significantly from that of traditional distributed file systems. First, it has to be simple and lightweight considering the limited CPU bandwidth and memory of sensor nodes. For example, MicaZ [23], one of the most current platforms, has only an 8MHz 8-bit processor and a 4KB RAM. Second, directories and file names need not be maintained in the sensor network itself. These abstractions are needed only on the collection station. Sensor nodes simply write data to the collection station (via a delay tolerant network). They never read the data they write. The system essentially abstracts away the delay-tolerant network between the data sources and the sink. Third, data redistribution is an essential component of the system to avoid inefficiencies of partitioned storage. A good data redistribution scheme is needed to migrate data from highly utilized to less utilized storage spaces in order to improve overall storage utilization. Finally, sensor nodes may not necessarily be connected, forming multiple network islands. Opportunistic data redistribution is needed not only within, but across such islands. These concerns are addressed in EnviroStore design.

We implemented EnviroStore in nesC under TinyOS [12] and evaluated system performance experimentally on TOSSIM [14]. Our evaluation results demonstrate the effectiveness of this service in improving storage utilization in various application scenarios. Up to an order of magnitude improvement was observed in postponing the onset of data loss.

The rest of the paper is organized as follows. Section II presents system design. Section III presents the details of EnviroStore implementation. Section IV illustrates evaluation results. Section V reviews related work. Section VI concludes the paper.

II. Design

A. System Model

We consider a sensor network that is normally disconnected from the outside world. The function of this network is to collect sensory data. These data must eventually be moved to a data sink. In our architecture, the sink is a process that runs on a user's PC, identified by a regular IP address and a (well-known) TCP port. This process implements a file service that receives sensor network data and organizes them in a local file system on the PC in accordance with some configuration setup. The sink typically has Internet access. Hence, data can be uploaded to it via the Internet from a remote network. Alternatively, the sink may have an 802.15.4 interface (or an 802.15.4 mote connected to the serial port). A process reading that interface (or serial port) relays data to the well-known port of the file service for storage.

Since the sink is normally disconnected from the sensor network, data must be buffered in the sensor network until an upload opportunity arises. Hence, a sensor network storage system is needed. We call this storage system *disruption-tolerant* since it should accommodate sensor network partitions. The disruption-tolerant storage system should, for example, be able to take advantage of data mules to share data across partitioned network islands or offload data to the sink, as is shown in Figure 1. Data mules are any (trusted) mobile devices that may come in contact with sensor network islands. For example, they might be handhelds with an 802.15.4 interface and an 802.11 interface. Depending on the application scenario, mules may be eventually able to contact the sink via the Internet or via the 802.15.4 interface.

In the context of sensor network applications that motivate this paper, only two types of data mules are relevant. The first type represents data mules that intentionally relay data between the sink and the sensor nodes. For example, network maintenance operators who visit the network periodically may also perform data mule functions. Barring unexpected failures, this type of mule is guaranteed to return the data to the sink. For example, it may have Internet access such that it will send collected data to the sink via the Internet when it encounters an access point upon return from the field.

The second type of mule is one whose mobility patterns are independent of data upload needs. For example, consider a library-monitoring study that measures noise levels in different rooms and correlates them with library use[2]. Due to the size of the rooms, not all noise sensors are connected. A librarian performing their normal job functions (that are independent of data collection) can carry a data mule device. Nodes that come in contact with the mule will then opportunistically use it for data upload or redistribution. Another example of independent mobility patterns is inspired by a recent experience of the authors, where sensor nodes deployed in a forested area were repeatedly visited by raccoons. The recurrence of these visits suggested the possibility of using members of the local wildlife as data mules. A similar observation was made when an experimental farm was considered for two concurrent telemetry experiments: one was to collect chemical measurements from soil; the other was to track cattle in the same farm using GPS collars. The natural opportunity to design the collar to perform data mule functions for soil sensors suggested a deeper point; as sensors proliferate, the role of opportunistic exploitation of natural mobility in the environment may become more important in network protocol design. Independent mobile data

[2] This is an actual study planned at the UIUC library to test the hypothesis that optimal use does not require a noise-free environment

Application	Sensor Nodes		Data	Data Sink	
	Static	Mobile	Mules	Static	Mobile
GDI [20][22]	√			√	
ZebraNet [17]		√			√
NIMS [1]	√	√			
Macroscope [24]	√			√	
Underwater Sensornet [26]	√		√		
Smart Attire [10]		√		√	

TABLE I
APPLICATION EXAMPLES

mules are opportunistically exploited by EnviroStore.

The disconnected operation models described above are observed in a large set of other sensor network applications including environmental monitoring, animal tracking, and assisted living. Table I lists some concrete examples from the recent literature and their suitability for this system model.

B. System Design

In this section, we present a set of mechanisms that maximize the effective storage space of the sensor network.

1) In-network Data Redistribution: As mentioned earlier, the distribution of the data inputs is not necessarily even among sensor nodes, which calls for data redistribution to improve total storage utilization. An ideal data redistribution scheme should accommodate all data as long as the sum of all node sensory inputs, accumulated over time, is less than the sum of all node storage capacities. A simple solution is to balance storage utilization by offloading data from nodes that are highly loaded to nodes that are not. In a perfectly balanced system, no storage overflow occurs until the total network capacity is exceeded when all nodes reach their flash limit simultaneously. This scenario represents optimal flash usage. Unfortunately, the solution is not energy-efficient. A small change in storage utilization of one node (due to new input) may result in data dissemination to every node in the network even if the source has plenty of storage to accommodate the input. This excessive and unnecessary communication may result in early energy depletion and consequent untimely loss of data.

From an energy saving perspective, it is therefore advantageous not to start offloading data too early. In other words, a *lazy-offload* scheme is preferred. By postponing data balancing until the latest possible time (when flash overflow is imminent), significant energy savings can be achieved. For efficiency reasons, we are interested in algorithms that use local information only. In such algorithms, plateaux must be avoided in the neighborhood to ensure that pathways exist for data flow. In accordance with the above two requirements, a node i, in EnviroStore, decides to offload data only when its remaining storage size (R_i) satisfies one of the following two conditions:

$$R_i = R_{min} \; and \; R_i < R_{TH} \tag{1}$$

$$R_i > R_{min} \; and \; R_i - R_{min} < R_{gradient} \tag{2}$$

where R_{min} is the minimum remaining storage size within node i's neighborhood (including i), R_{TH} is a configurable threshold to delay data transfer until the remaining free storage is small enough, and $R_{gradient}$ is a configurable parameter introduced to allow for a certain level of local imbalance whose gradient points in the direction of less utilized areas in the network. The first condition indicates that the most-loaded node does not start to transfer data until its remaining storage falls below a threshold R_{TH}. This is to prevent unnecessary energy consumption. R_{TH} should be set big enough to accommodate temporary bursts of input data. The second condition ensures that plateaux do not occur among neighbors, so that data can always flow away from congested nodes.

In our system, nodes exchange periodic advertisement messages sharing free storage information, R_i, within their neighborhood. To conserve energy, such messages are sent at a low frequency (e.g., once per minute). However, to ensure accurate knowledge of neighborhood data to within specified error bounds, extra advertisement messages are inserted when the remaining storage size incurs big changes (of more than $R_\triangle$, the *node advertisement threshold*) since the last advertisement.

If one of the above offload conditions is satisfied, node i should next decide who to send data to. Obviously, i should select the destination from those underloaded neighbors whose remaining storage size is above the average remaining storage ($\bar{R}_i$) of the neighborhood (including node i itself). Always selecting the neighbor with the largest remaining storage turns out to be a bad choice, because a node with the largest R within its neighborhood may be chosen simultaneously by multiple other nodes as their distribution destination. After redistribution, the node may become overloaded and must transfer some of the recently received data back to its neighbors, causing unnecessary consumption of both bandwidth and energy. We call this phenomenon *data ping-pong*. To avoid it, we apply a random function which assigns each underloaded neighbor a non-zero probability (proportional to its remaining storage) of being selected as the redistribution destination.

To further prevent data ping-pong, we should bound the amount of data transfer. Assuming that node i selects node j as the redistribution destination, the amount of data to be transferred from i to j, denoted by D_{ij}, has to satisfy the following condition (not to "overdo" the transfer and reverse the direction of imbalance):

$$R_j - R_\triangle - D_{ij} \geq \bar{R}_j \tag{3}$$

Note that, $R_\triangle$ is added to account for the inaccuracy in the estimation of R_j on node i, which is caused by the low-frequency of node advertisements. The inaccuracy is bounded by $R_\triangle$ (assuming no message loss) since, as previously stated, extra advertisement messages are inserted for changes exceeding $R_\triangle$. Following the same reasoning, after the data transfer, the resulting free storage size of node i should not exceed the neighborhood average, since it may cause data ping-pong as well. Thus, we have:

$$R_i + D_{ij} \leq \bar{R}_i \tag{4}$$

Consequently:

$$D_{ij} \leq min(R_j - \bar{R}_j - R_\triangle, \bar{R}_i - R_i) \tag{5}$$

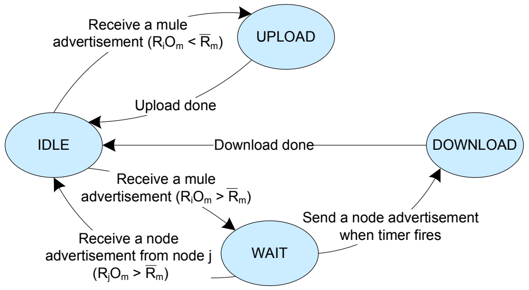


Fig. 2. State transition of a sensor node in cross-partition data redistribution

which is used to calculate the maximum allowable size of data transfer. The calculation is based on merely local information obtained through the aforementioned advertisement messages. EnviroStore then fetches a small chunk of data (i.e., a log item as defined in Section III) from local storage, and, if its size is below the maximum allowable size, reliably transfers it to the redistribution target. Data chunks are transfered until the conditions specified by Inequality (1) and Inequality (2) are invalidated.

Besides flash overflow, an important factor that may cause data loss is depletion of node energy. When a node runs out of energy, the data stored at the node can still be recovered by collecting the deployed node. However, the node obviously ceases to observe the environment, losing all subsequent measurements. To avoid such data loss, besides monitoring remaining free storage, our algorithm also keeps track of the remaining node energy (by reading the current voltage level and converting it to energy based on battery characteristics). A node i should not invoke or accept data redistribution (which spends extra energy to send or receive data) unless its estimated energy lifetime is longer than its estimated storage lifetime. This leads to:

$$\frac{\Omega_i}{E_i} > \frac{R_i}{S_i} \tag{6}$$

where Ω_i is the remaining energy, E_i is the initial energy, and S_i is the initial storage size.

2) Cross-partition Data Redistribution: The in-network data redistribution scheme works well when all sensor nodes are connected into a single network. However, in some application scenarios like the GDI deployment [22], sensor nodes are naturally deployed into network islands that can not communicate with each other. Even for an initially connected sensor network, practical issues like the instability of wireless channels, hardware or software failures, or depleted batteries may separate the network into islands. In such circumstances, it becomes critical to offload data from overloaded network partitions (in terms of storage capacity) to either data sinks (if present) or underloaded partitions. This is accomplished via mobile data mules.

We consider two types of (authenticated) mules. The first always carries the data back to the basestation. A node encountering such a mule can upload all its data to it. The second type is one whose mobility patterns are dictated by factors external to the storage system. The library example presented earlier is one such case. Such data mules can carry data to the basestation if they happen to come in contact with it. They can also be used for data redistribution across network partitions.

To identify nodes as overloaded or underloaded for redistribution purposes, a data mule must have a notion of a global average storage use. Accurately calculating the global average is virtually impossible, considering that the sensor nodes in different partitions can not directly communicate. Instead, each data mule m remembers the free storage advertised by each visited node. It uses their average $\bar{R}$ as an approximation of the global average. It then computes its own advertised free storage value $\bar{R}_m$ as the weighted sum $\alpha\bar{R} + (1-\alpha)R_m$, where R_m is the storage available on the mule itself. The parameter α of the mule is used to favor data redistribution versus upload. If α is close to 1, the mule favors redistribution to the neighborhood regardless of the storage available on the mule itself. If α is close to zero, it emphasizes upload, regardless of free storage available in the sensor network.

When mules have large storage or encounter the base frequently, to relieve the network storage, they should aggressively download data from the sensor nodes. To take this factor into account, a node i uses a weighted value R_iO_m (O_m is the occupancy ratio of the data mule m defined as the fraction of its local storage that is utilized) instead of its original remaining storage R_i to compare with $\bar{R}_m$. Obviously, the policy makes a mule download data more aggressively from sensor nodes when its occupancy is low (either because it has large unused storage or because it has offloaded most data to data sinks). Therefore, the cross-partition redistribution scheme can dynamically adapt itself to the differences in the size of storage space at mules as well as adapt itself to the visit frequency of data sinks if they exist.

Like sensor nodes, data mules also periodically advertise themselves by messages, but with a much higher frequency (e.g., once per second). Frequent mule advertisement is necessary for overloaded nodes to detect nearby mules as soon as possible in order to make the best use of data upload opportunities, as is shown in Figure 2. Frequent mule advertisement is feasible in terms of energy consumption since mules can be recharged frequently to obtain sufficient power.

At the same time, a mule should be able to detect nearby underloaded nodes to offload some of its data. The great difference between node advertisement frequency and mule advertisement frequency makes it much harder and slower for mules to detect nodes. To solve this problem, underloaded nodes, after receiving a mule advertisement, respond with a node advertisement to shorten the delay. These nodes use back-off timers (proportional to their current occupancy ratios) to suppress each other's node advertisement messages, as shown in Figure 2.

III. Implementation

We implemented EnviroStore using nesC in TinyOS. The implementation consists of three versions of code, encoding separately the subsystems run at sensor nodes, at data mules,

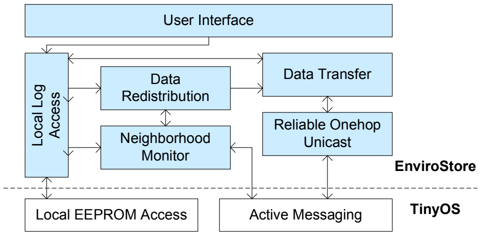


Fig. 3. System architecture for sensor nodes

and at data sinks. The overall system architecture for sensor nodes is depicted in Figure 3.

Application programmers use the service interface on the motes to submit data to be logged. Before data are written into local flash, such data are structured into the standard format of a *log item*, the minimum accessible data unit in EnviroStore whose data structure is depicted in Figure 4. Both writing and reading log items are functionalities supported by the *local log access* module. The *neighborhood monitor* module is responsible for sending advertisement messages within local neighborhoods. It also maintains a neighbor table to keep track of the storage status of each neighbor. At the same time, it is responsible for detecting mules via the reception of mule advertisement messages. Based on the conditions described in Section II-B.1 and II-B.2, the *data redistribution* model determines whether the current node should offload data to it neighboring nodes or the detected mule, and, if so, it calculates the maximum amount of data transfer and signals the *data transfer* module to start data transfer towards a selected neighbor or the nearby mule. The *reliable one-hop unicast* module, as its name suggests, provides reliable unicast for nodes to transfer log items. To avoid data loss during transfer, the *data transfer* module never deletes a log item until the *reliable one-hop unicast* module acknowledges its reception.

The implementation for data mules has a similar set of modules except that the neighborhood monitor module records all the nodes a mule has met (in other works, dynamic neighbors of the mule) rather than a static set of neighboring nodes.

A. Local Storage Structure

The local storage space of nodes is organized into a circular buffer containing continuous log items (Figure 4). The *head* points to the next log item to be read or deleted, and *tail* pointer points to the next position to write a new log item. This simple data structure proves to be very suitable for current sensor network platforms. First, it meets all the requirements of EnviroStore since the system model suggests that random access to the logs is not required since we are not targeted for runtime data acquisition. Second, it consumes minimum code and data memory as this data structure organizes occupied space into a continuous data chunk, eliminating the need for any complex space management mechanisms like free space management or defragmentation. Third, it may prolong flash lifetime by balancing write access to different locations. Note that the endurance of the 512 KB serial flash on most current TinyOS platforms is only 10,000 erase/write cycles.

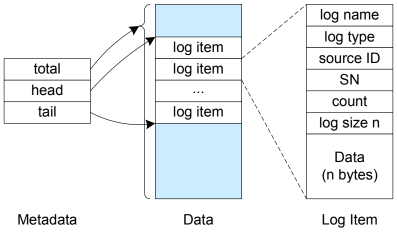


Fig. 4. Local storage structure

The sequential write access to flash of the circular buffer structure guarantees that the number of writes to different flash blocks is almost perfectly balanced with a maximum difference of one.

B. User Interface

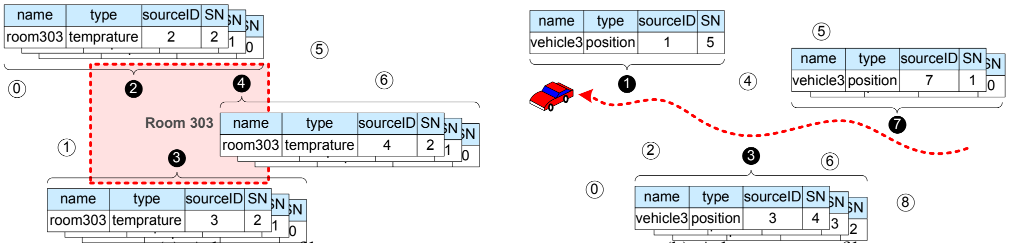
EnviroStore supports two types of log files as Figure 5 depicts, and provides two nesC commands for them, respectively. The first type of log file is simultaneously written by different nodes, with each node generating a sequence of log items with continuous serial numbers. All the log items from multiple nodes form an array of log sequences. Therefore, we call this type of log file *log-array files*. Log-array files are useful for logging attributes of an environmental event that is independently monitored by multiple nodes, for example, to obtain the temporal and spacial distribution of the temperature in Room 303 as shown in Figure 5(a).

The other type, named *log-sequence files*, expects one writer at a time. Multiple nodes should coordinate with each other so that the next writer does not start before the previous one stops. Unique and continuous serial numbers must be used. This mode is developed for compatibility with EnviroSuite [18], a middleware service that tracks mobile environmental entities (such as vehicles). The service elects a unique leader node in the vicinity of the tracked entity and hands off leadership from node to node as the entity moves. The leader maintains a unique ID. This ID can be used as the log name to produce a distributed log-sequence file that stores the history of a target along its trajectory. An example is shown in Figure 5(b), where EnviroSuite associated an ID (`vehicle3`) to a vehicle, and elected nodes 7, 3 and 1 sequentially to log the vehicle's current state. The resulting log-sequence file, using `vehicle3` as its name, contains the trajectory of the vehicle.

IV. Evaluation

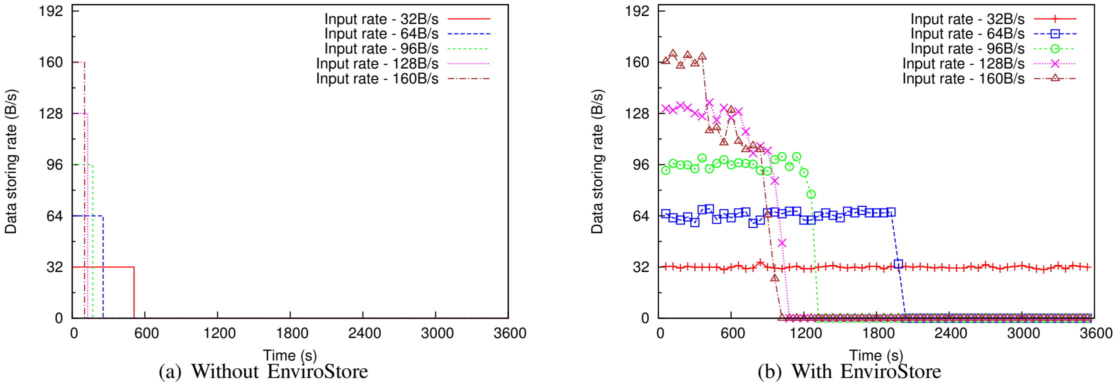
This section presents a performance evaluation of EnviroStore. EnviroStore is implemented in nesC on TinyOS. We use TOSSIM that provides a high fidelity simulation of TinyOS applications, precisely modeling the 40Kb network at the bit level and the CPU clock at a 4MHz granularity.

Figure 6 depicts the basic deployment configuration used throughout the evaluation. On a 80×80 ft^2 field, we deploy 36 nodes (circles labeled with node IDs) into four network partitions. Nodes in black (node 4 and 31) are data generators. They run an application that periodically creates input for EnviroStore. An unbalanced deployment configuration is used to stress EnviroStore. In the top-left and the bottom-right network partitions, only one node generates all the input, the



(a) A log-array file (b) A log-sequence file

Fig. 5. Examples of different types of log files



(a) Without EnviroStore (b) With EnviroStore

Fig. 7. Data storing rate at different time

others being silent. The other two partitions do not have input to further stress the redistribution algorithm.

When present, a basestation is placed at the position marked by ×, which is also the starting point of data mules. The movements of the mules follow a constraint random walk model, moving 5ft every second and turning a random angle between $-\frac{\pi}{6}$ and $\frac{\pi}{6}$. The random walk serves our purpose well because EnviroStore has no knowledge about mobility of mules.

The final simulations of our experiments are very heavy-weight, especially when mules are used, mainly because of frequent channel update in TOSSIM caused by the mobility. For a network of 36 nodes, it takes 6-10 hours of wall-clock time on a Pentium4 1.7GHz machine with 1GM RAM to simulate 3600 seconds of virtual time. To accelerate the evaluation, we set storage capacity of the devices to be smaller than that of current hardware platforms. The storage of the node (S) is set to 16KB. The mules have a relatively larger storage of 64KB. Consequently, while the *absolute time* when the system encounters data loss will not match real platforms, relative performance of different algorithm will remain the same. Hence, inferences can be made about multiplicative improvement factors over a baseline.

Unless otherwise indicated, R_{TH}, $R_{gradient}$, and $R_{\triangle}$ are set to be $0.95S$, $0.05S$, and $0.01S$ (S is the total storage of a node), respectively. Recall that R_{TH} has to be big enough to accommodate bursts of input. We use a large percentage of the storage as R_{TH} since the total storage size is small. Next, we investigate a disconnected sensor network with and without partitions.

A. Scenario 1: Single Disconnected Sensor Network

In a single disconnected sensor network, the network is not partitioned. Also, neither mules nor a basestation is present. For this scenario, we use only the top-left partition shown in Figure 6.

To illustrate how EnviroStore maximizes storage capacity via in-network data redistribution, Figure 7 compares the data

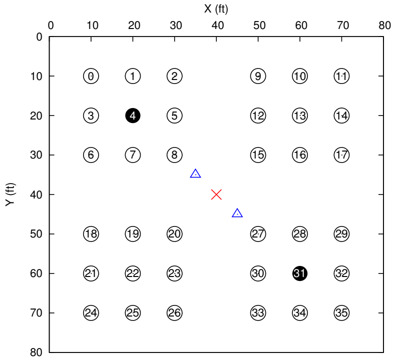


Fig. 6. Basic deployment configuration

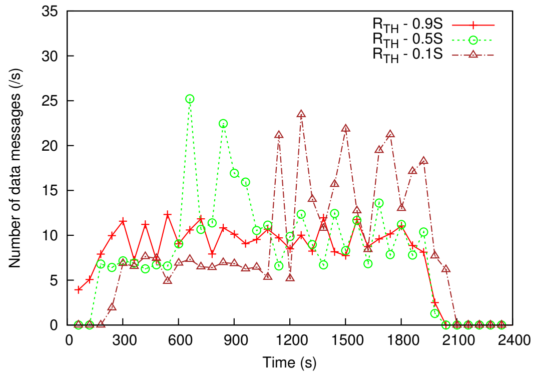


Fig. 8. Comparison of data storing rate at different time

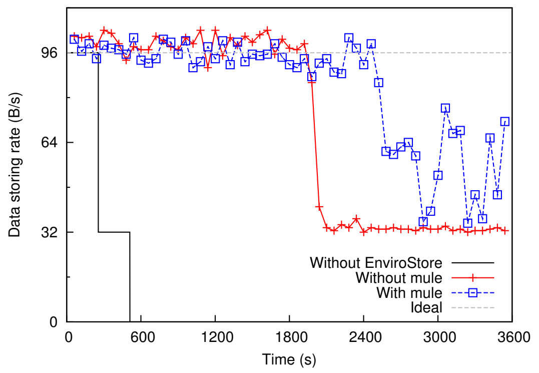


Fig. 9. Comparison of data storing rate at different time

storage rates (i.e., the amount of data stored by EnviroStore per second) for different input rates with and without EnviroStore. If storage is infinite on each node, the data storage rate always equals the input data rate. However, data loss caused by insufficient local storage makes the data storage rate drop below the input data rate. As can be seen in Figure 7, for all the input rates, applying EnviroStore significantly delays data loss. For example, with an input rate of 64B/s, the first appearance of data loss is delayed from time 256s to 1900s (i.e., more than 6 times).

Notice that for input rates lower than 96B/s, the corresponding data storage rates stay close to the input rates until they sharply drop to zero. In contrast, for input rates higher than 96B/s, the data storage rates decline *gradually*. The underlying reason is that when the input rate exceeds a node's communication bandwidth, new data arrives before the data redistribution algorithm converges to a balanced state. When the data generator (node 4) completely consumes its local storage, EnviroStore redistributes at the communication rate, which is below the input rate, yet above zero.

To investigate the effects of R_{TH} on the data storage rate and energy consumption, we fix the input rate to be 64B/s and use different values of R_{TH} ($0.9S$, $0.5S$, and $0.1S$). Using a smaller R_{TH} does not have an appreciable impact on the data storing rate. However, as expected, it does affect energy consumption, as shown in Figure 8 which plots the number of data messages sent per second for different values of R_{TH}. As can be seen, setting R_{TH} to $0.5S$ postpones extensive data transfer from 180s ($0.9S$) to 540s. Using $0.1S$ as R_{TH} further postpones it to 1200s. If the application input happens to stop at 1200s, setting R_{TH} to $0.1S$, comparing to $0.9S$, yields significant energy savings due to lazy offload.

B. Scenario 2: Partitioned Sensor Network with Data Mules

In order to analyze the effects of cross-partition redistribution, we deploy four network partitions consisting of 36 nodes as shown in Figure 6, as well as one mobile data mule. The input rates at node 4 and node 31 are set to be 64B/s and 32B/s, respectively. This setting provides the four partitions with three levels of input rates: 64B/s for the top-left one, 32B/s for the bottom-right one and 0B/s for the others. Great differences in input rates stress the distribution algorithm.

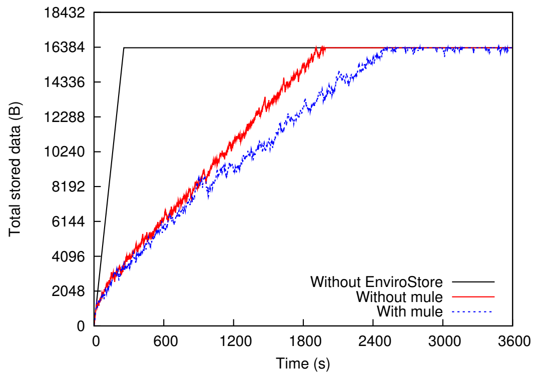
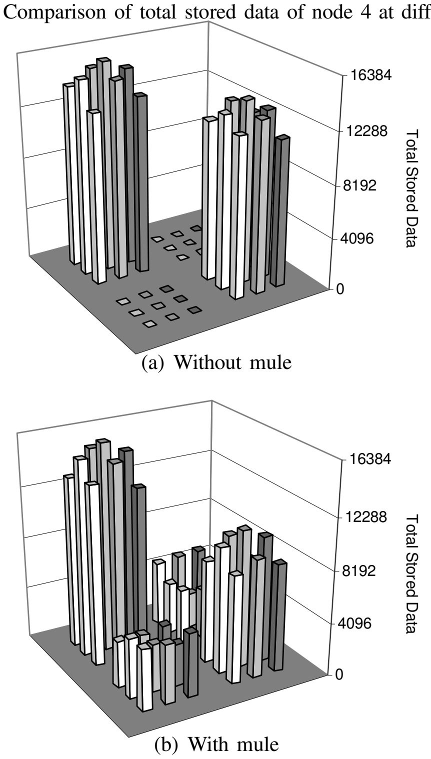


Fig. 10. Comparison of total stored data of node 4 at different time



(a) Without mule

(b) With mule

Fig. 11. Distribution of total stored data after one hour

Figure 9 presents the data storing rate of different configurations. Ideally, the data storage rate should always equal to the input rate (96B/s). For the current configuration, without EnviroStore, the data storage rate drops from 96B/s to 32B/s after 256s when node 4's storage is exhausted, and to 0 after 512s when node 31's storage is exhausted. After applying in-network redistribution (without mules), data loss does not occur until after about 1900s. If we further invoke cross-partition redistribution by introducing a mule, the data storage rate stays at 96B/s until after 2400s. Overall, applying EnviroStore delays data loss by a factor larger than 8.

By carrying data from overloaded network partitions to underloaded partitions, the data mule delays the time that the storage of the most-loaded partition (the top-left one) gets exhausted. Figure 10 demonstrates this effect by showing the total stored data at node 4 over time. Obviously, in-network redistribution (without mules) reduces the storage consumption speed at node 4. Cross-partition redistribution (with a mule) further slows down the storage depletion.

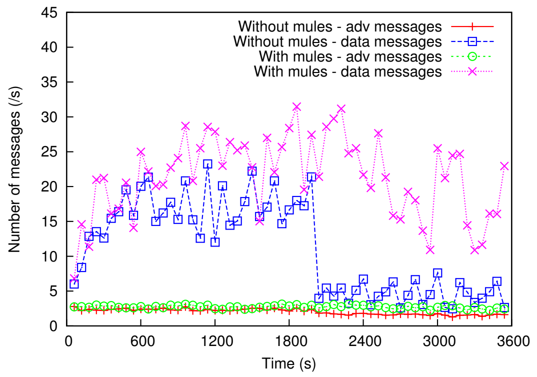


Fig. 12. Comparison of number of messages per second at different time

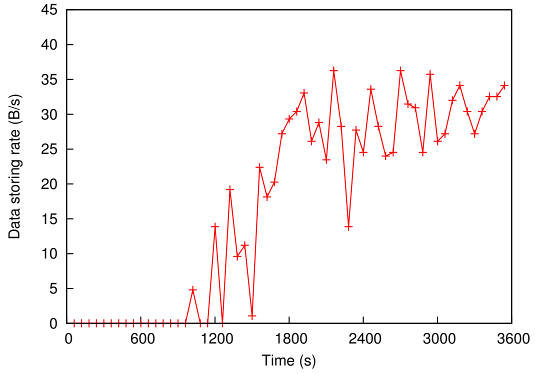


Fig. 13. Data storing rate of the basestation over time

Figure 11 illustrates the distribution of total stored data among different nodes after 3600s of virtual time for deployment configurations without and with the mule. Obviously, the mule successfully moves data from overloaded network partitions to underloaded partitions and achieves a more balanced storage occupancy.

We also explore the energy overhead (in terms of messages) of redistributing data to maximize effective storage. The number of advertisement messages and data messages sent per unit time by the sensor nodes for both configurations with and without the mule are plotted in Figure 12. As shown, the total number of messages sent per second for the whole network is always below 36, in other words, below 1 per node, which is acceptable for sensor networks. For the configuration without a mule, after around 1900s, the number of data messages per second drops to 5 abruptly and the number of advertisement messages decreases as well. This is when the top-left network partition gets exhausted and stops accepting input as well as data redistribution, which is consistent with what we observe in Figure 9. We do not see such a sudden drop in the configuration with the mule because (i) prior to time 2400s the top-left partition is not full, and (ii) after time 2400s the mule is still actively communicating with the nodes to do cross-network redistribution.

Finally, we add a basestation, marked by $\times$, and two extra nodes, marked by $\triangle$, in Figure 6). They ensure connectivity between the sensor nodes and the basestation. Figure 13 shows the data storage rate of the basestation over time. Recall that we allow certain storage imbalance between nodes by using $R_{gradient}$ of $0.05S$. Therefore, the data storage rate of the basestation stays zero for a certain duration, then increases gradually.

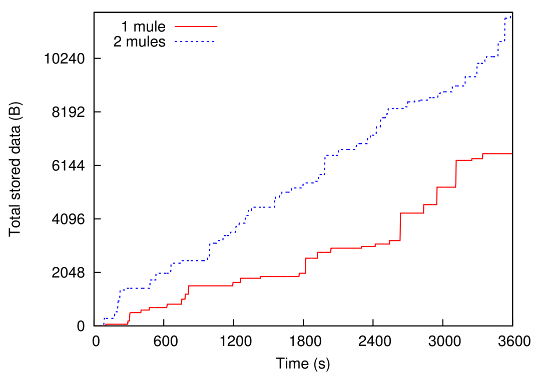


Fig. 14. Total stored data of the basestation over time

In the next experiment, we remove the two nodes marked by $\triangle$ in Figure 6 to disconnect the network partitions from the basestation and add one or two data mules to see how they help with data upload. Figure 14 depicts the amount of accumulated data at the basestation over time. Having more mules increases the rate of uploading data to the basestation, as shown in Figure 14.

In summary, the evaluation results demonstrate that EnviroStore is applicable to a wide set of application scenarios. It effectively maximizes the network storage capacity through in-network and cross-partition data redistribution.

V. Related Work

The problem of data storage on individual nodes has been addressed adequately in previous work such as ELF [5], a log-structured and flash-based file system, and Matchbox [25], a simple file system distributed with TinyOS. More recently, several distributed storage services have also been proposed. One example is [7], a two-tier data-centric storage and retrieval service using distributed hash table and double-ruling. TSAR [6] and PRESTO [15] also feature a two-tier data storage architecture comprising sensor nodes and proxies for data acquisition and query processing. DIMENSIONS [9] is another system that is designed to store long-term information by constructing summaries at different spatial resolutions using various compression techniques. TinyDB [19] and related projects [2][29] organize sensor networks and their collected data as a distributed database and focus on query processing techniques to acquire data from such databases. All these services assume connected operation and real-time data acquisition. Geared for disconnected operation, EnviroStore has a completely different focus. Namely, it investigates cooperation between different tiers (sensor nodes, data mules, and base stations) to maximize storage capacity.

One key challenge in EnviroStore is data redistribution. The sensor network community has applied the balancing techniques for other purposes, including maximizing system lifetime by balancing energy consumption of different nodes [16], and improving fairness by balancing MAC layer accesses [28]. Data redistribution in EnviroStore bears some similarity with the former. However, we have the additional control knob of exchanging data between nodes, while it is not possible for nodes to charge each other using their own batteries.

More broadly, load-balancing comprises many algorithms that are studied in different application contexts. Representative applications include load-balancing in web servers [3][27][30], P2P networks [11][13], wireless LANs [8], and distributed operations systems [4]. These applications commonly involve many nodes, which can range from web servers to P2P clients, each with a finite resource capacity. The particular resource may be bandwidth, computing power, or storage space. When more resources than desired are consumed, a node tries to reduce its resource consumption by transferring some load to its peers. While this general description also applies to EnviroStore, EnviroStore is considerably different. First, EnviroStore has the extra constraint of limited energy, which leads to new insights such as lazy offload. Second, because of the resource limitations of individual nodes, no single node is able to coordinate with all the other nodes. Load balancing in EnviroStore must be completely distributed, dependent only on local information. Third, EnviroSuite has the additional challenge of redistributing data between entities that are disconnected.

VI. Conclusion

In this paper, we presented EnviroStore, a cooperative storage system that maximizes network storage capacity in the presence of disconnected operation in wireless sensor networks using in-network and cross-partition data redistribution mechanisms. Our evaluation study validates that EnviroStore can (i) effectively utilize the network storage capacity of disconnected sensor networks to accommodate the most sensory data, and (ii) opportunistically offload data from overloaded network partitions to underloaded partitions via mules. Moreover, the system model based upon which we design EnviroStore is flexible enough to allow a wide set of applications to take the best advantage of EnviroStore. This paper serves as our initial attempt to design a storage system for disconnected operation. We plan to further extend the work in several directions. These directions include (i) use of controllable data mules to optimize data redistribution and upload, (ii) investigation of data replacement policies to maximize the total amount of valuable information instead of just the amount of stored data, and (iii) performance evaluation of EnviroStore on real hardware platforms.

References

[1] M. A. Batalin, M. Rahimi, Y. Yu, D. Liu, A. Kansal, G. S. Sukhatme, W. J. Kaiser, M. Hansen, G. J. Pottie, M. Srivastava, and D. Estrin. Call and response: experiments in sampling the environment. In *SenSys*, 2004.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *MDM*, 2001.

[3] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. In *IEEE Internet Computing*, 1999.

[4] M. Correa, A. Zorzo, and R. Scheer. Operating system multilevel load balancing. In *SAC*, 2006.

[5] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *Sensys*, 2004.

[6] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: A two tier sensor storage architecture using interval skip graphs. In *Sensys*, 2005.

[7] Q. Fang, J. Gao, and L. Guibas. Landmark-based information storage and retrieval in sensor networks. In *Infocom*, 2006.

[8] gal Bejerano and S.-J. Han. Cell breathing techniques for balancing the access point load in wireless lans. In *Infocom*, 2006.

[9] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. S. Heidemann. An evaluation of multi-resolution storage for sensor networks. In *Sensys*, 2003.

[10] R. K. Ganti, P. Jayachandran, T. F. Abdelzaher, and J. A. Stankovic. Satire: a software architecture for smart attire. In *MobiSys*, 2006.

[11] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *Infocom*, 2004.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX*, 2000.

[13] K. Kenthapadi and G. S. Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *SPAA*, 2005.

[14] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: Accurate and scalable simulation of entire tinyos applications. In *SenSys*, 2003.

[15] M. Li, D. Ganesan, and P. Shenoy. Presto: feedback-driven data management in sensor networks. In *NSDI*, 2006.

[16] Q. Li, J. Aslam, and D. Rus. Online power-aware routing in wireless ad-hoc networks. In *Mobicom*, 2001.

[17] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *MobiSys*, 2004.

[18] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. In *ACM Transactions on Embedded Computing Systems*, 2006.

[19] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 2005.

[20] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, 2002.

[21] R. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: Modeling a three-tier architecture for sparse sensor networks. In *SNPA*, May 2003.

[22] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *SenSys*, 2004.

[23] TinyOS platforms. *http://www.tinyos.net/scoop/special/hardware/*.

[24] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *SenSys*, 2005.

[25] U. C. Berkeley. Tinyos, 2005. `http://www.tinyos.net/`.

[26] I. Vasilescu, K. Kotay, D. Rus, M. Dunbabin, and P. Corke. Data collection, storage, and retrieval with an underwater sensor network. In *SenSys*, 2005.

[27] J. L. Wolf and P. S. Yu. Load balancing for clustered web farms. *SIGMETRICS Perform. Eval. Rev.*, 2001.

[28] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MOBICOM*, 2001.

[29] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 2002.

[30] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Trans. Parallel Distrib. Syst.*, 2005.