DALi: A Communication-Centric Data Abstraction Layer for Energy-Constrained Devices in Mobile Sensor Networks

Christopher M. Sadler and Margaret Martonosi

Department of Electrical Engineering
Princeton University
{csadler, mrm}@princeton.edu

ABSTRACT

Communications in mobile and frequently-disconnected sensor networks are characterized by low-bandwidth radios, unreliable links, and disproportionately high energy costs compared to other system operations. Therefore, we must use as efficiently as possible any periods of connectivity that we have. For this reason, nodes in these networks need mechanisms that organize data to streamline search operations, local computation, and communications.

This work proposes a Data Abstraction Layer (DALi), which is inserted between the application layer and the file system. DALi organizes data with networking in mind to facilitate the development of services for Data Search, Naming, and Reduction that combine to make communications more efficient. From the resulting two-tiered data hierarchy, we develop a multi-layer drill-down search structure that can locate data multiple orders of magnitude faster (and with much lower energy) than simpler data storage structures. Additionally, DALi conserves energy and bandwidth through a mechanism that acknowledges and removes specific data segments from a mobile sensor network. Finally, it seamlessly integrates in a lossless compression algorithm specifically designed for sensor networks to save additional energy.

Categories and Subject Descriptors: C.2.1 [Network Architecture and Design]: Wireless communication; C.3 [Special-Purpose and Application-Based Systems]: Real-time and embedded systems; D.4.3 [File Systems Management]: File Organization; E.5 [Files]: Sorting/Searching

General Terms: Algorithms, Management, Performance

Keywords: Data Search and Storage, Energy Efficient Communications, Mobile Ad Hoc Sensor Networks

1. INTRODUCTION

Mobile and frequently disconnected sensor networks form an interesting subset of the sensor network design space. The target applications vary drastically, from zebra tracking [39] to polar monitoring [4]. They have an underlying set of common traits, however, largely based on their harsh operating environments which can make physical node access difficult and which place logistical limits on the size of the deployments.

These systems are characterized by both the severe resource constraints of sensor nodes and by short periods of unreliable, low quality communications over low bandwidth radios. Beyond their sensors, they collect and store data using an ultra-low power microcontroller and energy-efficient, non-volatile memory in an effort to operate for months at a time on a limited energy budget.

Over time, sensor node microcontrollers have become more capable, the amount of storage space has increased, and the energy costs of CPU and storage have decreased. These trends are likely to continue. However, radio transmissions have remained expensive and unreliable and this is unlikely to improve significantly over time. Significant challenges exist regarding the physical energy costs of wireless signal propagation, the difficulties of designing appropriate antennas, and environmental factors which are exacerbated by a constantly changing network topology. Additionally, in a mobile network, nodes may transmit multiple replicated copies of the data to balance latency and energy constraints [32][34]; unnecessarily transmitting the data either to nodes that already have it or to anyone after the sink has received a copy wastes valuable bandwidth and energy.

As a result, a good mobile sensor system must be designed with the data storage and communication infrastructure in mind. Current Flash file systems designed for stationary sensor networks offer clear advantages over raw application management of data, but on their own these systems do not meet our goals. For example, files can grow to the size of the Flash, there is no efficient way to identify particular data items in files, and there is minimal support for compression. However, for the tasks for which they were intended, such as using the Flash efficiently and ensuring data integrity, these file systems perform well. For this reason, we have developed DALi, a Data Abstraction Layer for mobile sensor networks that lies between the application and the file system and provides nodes with Data Search, Naming, and Reduction services.

Data Search is the ability to quickly locate specific data on the node, by name or by value, and summarize it when appropriate. We emphasize search speed because minimizing query response times improves bandwidth efficiency.

Data Naming is the ability to identify specific sections of data in a granularity that can be easily transmitted through

MobiSys'07, June 11-14, 2007, San Juan, Puerto Rico, USA.

the network. Using this, we can build "delete lists" which aim to conserve energy and bandwidth by stopping data (once delivered to the sink) from being unnecessarily transmitted further. We can likewise prevent nodes from transmitting data to other nodes that already have it.

Data Reduction is the ability to shrink the data through in-network computation, data aggregation, or compression. This mechanism conserves energy and bandwidth by reducing the volume of data in the network.

These three functions are interdependent and a truly effective system for mobile sensor networks needs to provide services for all three.

The contributions of this work include:

- We design and develop a prototype of a Data Abstraction Layer (DALi) that restructures data in a way that simplifies communications and uniquely incorporates each of the processes of Data Search, Naming, and Reduction.
- We introduce an efficient way to incorporate "delete lists" into the system, which can reduce energy consumption by multiple orders of magnitude by reducing unnecessary transmissions.
- We demonstrate that our hierarchical data organization serves as the basis for a drill-down search structure that allows for simple, fast sensor data searches on both spatial and temporal data. DALi can effectively search large real-world datasets in the amount of time it takes to send a handful of packets.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 then presents an overview of DALi. Section 4 introduces DALi's Data Search service and Section 5 introduces its Data Naming and Data Reduction services. Then, Section 6 offers a discussion and evaluation of our work, and Section 7 concludes the paper.

2. RELATED WORK

Sensor network data management systems are typically tailored to stationary, well-connected sensor networks and, therefore, do not attempt to leverage Flash data storage to make opportunistic mobile communications more efficient. Additionally, no existing data organization offers a combination of Data Search, Naming, and Reduction services similar to those offered by DALi.

Proactive relational query processors [11, 13, 19, 38] use queries to activate specific sensors on a node, collect readings for a given period of time, and return the results. They may also compute data summaries and aggregate data from multiple nodes. However, mobile delay-tolerant networks must be reactive because of the long latency required to deliver queries to nodes. Additionally, the often-changing network topology prevents nodes from employing aggregation techniques that rely on data correlation in the network or distributed schemes that rely on specific sensors to execute specialized data reduction algorithms; nodes should primarily rely on data reduction algorithms which can be executed locally. For stationary sensor networks, both more traditional [30] and distributed [24] storage abstractions exist as well, but frequent disconnections and the sparse distribution of nodes prohibit us from using these methods.

Generic Flash file systems [1, 35] and Flash file systems for sensor networks [6, 7, 10] store data in arbitrarily large files, like PC-based file systems. These files can be far larger than mobile nodes can transmit in one communications period. The file systems have no mechanisms for identifying smaller segments of the file, which is critical to preventing unnecessary communications related to duplicate copies of data in the network. Our work provides the additional services necessary for proper data location and identification to assist communications.

Our mechanism for subdividing data into smaller chunks is similar to the BitTorrent peer-to-peer file distribution system [5]. However, existing variants for MANETs [23] and sensors [33] are inappropriate for our networks because each node uses a tracker to find out which peers have the file it wants—information that is not likely to be available—and assume good connectivity and reliable multi-hop routes through the network that can move large volumes of data at once. DALi, on the other hand, gradually acquires data over the independent, opportunistic peer-to-peer links characteristic of mobile sensors.

Our data division mechanism also resembles the SPIN routing protocol [12, 15], which, in simulation, breaks data into 500B segments and uniquely names them in an effort to suppress redundant transmissions. However, SPIN requires that each application provide its own naming scheme. DALi, on the other hand, provides a standard two-level naming structure which is applicable across applications and allows names to be merged so that one name can represent much more data. It also provides search and data reduction services that SPIN does not consider.

Other Flash-based sensor storage systems offer useful data structures (e.g., Capsule [20]) and search capabilities (e.g., MicroHash [16]) and are a strong influence for our work. However, they do not attempt to tackle the issue of simplifying communications and they were never intended to transmit more than short data summaries, directly related to their intended use on stationary, connected networks. Additionally, the search algorithms for systems such as MicroHash are only designed to handle data collected from a single node and will not work if the data is not stored in time order. Unordered data is common in systems that store and process data from multiple nodes like DALi does.

Ganesan et. al. use wavelet summarization both on a single node and over groups of nodes to offer multiple granularities of data for transmission and search [9]. Their concept of drill-down queries is similar to ours, but we generate metadata rather than wavelets since they are not appropriate for answering queries on all types of data. Additionally, we cannot expect to have enough nodes, the proper node topology, or the data correlation necessary for wavelets to be effective across groups of nodes.

Delay-Tolerant Networks [8] may use data mules to gather data files from stationary sensor nodes [26]. However, communications between the sensing nodes and the data mule may be unpredictable, unreliable, and intermittent, especially if the mule's movements are random. DALi can assist data delivery in these scenarios by dividing files into more communicable segments.

3. DALI ARCHITECTURE

Nodes in sparse, mobile sensor networks will often adapt between either sending all of their data, which minimizes latency, or responding to specific queries, which minimizes communications. If the application wants all nodes to send all data, nodes need intelligent ways to prevent costly unnecessary communications and to improve the efficiency of

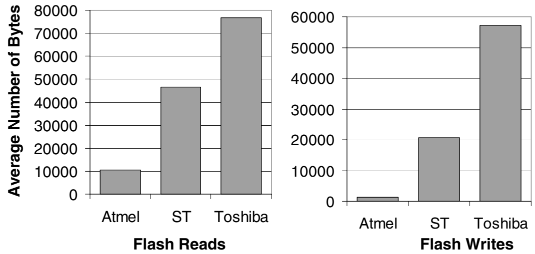

Figure 1: Average number of bytes that a node can read from and write to Flash (assuming page sized operations) for the same amount of energy as transmitting one byte over the XTend radio at 500mW.

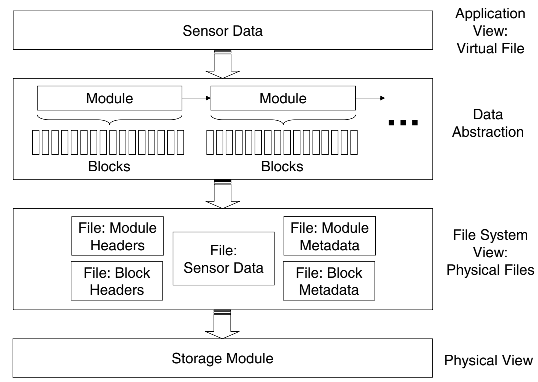

Figure 2: DALi's Architecture.

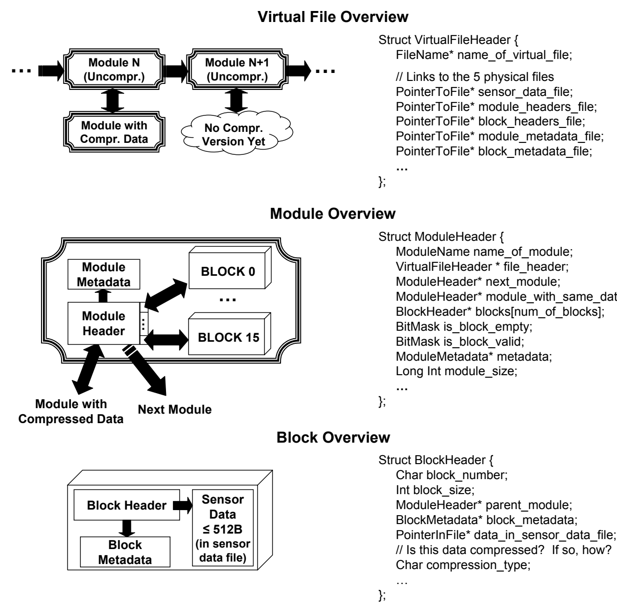

Figure 3: Left: Abstract view of how data is stored in Virtual Files, Modules, and Blocks. Right: Pseudo-code structure definitions of Virtual Files, Modules, and Blocks. Only selected fields are listed for each structure. Although this code depicts a number of memory consuming pointers, the actual implementation is designed in a way that minimizes RAM usage.

necessary ones. If the application just wants answers to specific questions, nodes need to quickly query and summarize a subset of the data that they have collected.

DALi provides these mechanisms on resource constrained sensor nodes and, in the process, provides a semblance of communications reliability in an unreliable realm. These mechanisms should be applicable to both data collected locally and to data received from other nodes, so that a node can preemptively answer data requests for its peers when possible.

An important point in this design discussion is that we are not deeply concerned about storage energy in mobile and frequently disconnected networks. As first pointed out in [21], the energy profile of the Flash has improved significantly over time. Figure 1 compares the energy cost of reading a byte from Flash and writing a byte to Flash with the energy cost of transmitting one byte over the XTend radio [22] at 500mW, as used in the ZebraNet project [39][1]. The comparison covers three Flash modules: a 4Mbit Atmel module [3] used in numerous prior sensor deployments (including ZebraNet), a newer 8Mbit ST module [27] which has many technological similarities to the Atmel Flash and is being incorporated into some newer sensor deployments, and a 1Gbit Toshiba module [31] which has been used in some recent sensor research [16, 20] and is likely to be incorporated into sensor deployments in the near future. For the Toshiba Flash, this translates to writing close to 3.7 million bytes for the energy required to transmit one 64B packet.

These trends suggest that we should use the Flash to our advantage as a way to improve the efficiency of our communications. DALi does this by reorganizing the data as it is written by the application; however, this should be done in a way that keeps the application interface simple.

3.1 DALi Structural Overview

DALi's architecture is shown in Figure 2 and an abstract view of the data organization is shown in Figure 3. The application creates a file, which we call a virtual file, in which it stores collected sensor data. This structure provides the application with a familiar file interface. In turn, DALi creates multiple physical files which combine to store headers and metadata used to identify and search the data, as well as the data itself (which is stored in a single physical file).

DALi breaks the data collected by the application into *Modules*, which are further subdivided into *Blocks*, a hierarchical design which was influenced by the structure of software updates in the Impala Middleware System [18]. Modules are designed to be easy to identify and Blocks are designed to pack data in small, trackable chunks. This structure simplifies the processes of delivering data to the sink and acknowledging that it arrived (see Section 5.1).

Modules consist of groups of 16 Blocks, enabling us to index any Block or groups of Blocks in a Module with a 16 bit mask. DALi uses a Block size of 512B so that the data fits in well with our sensor compression algorithm described in Section 5.2; it fills the Block with sensor readings until it is as close to 512B as possible without exceeding that amount. We define a fixed Block size rather than allowing it to vary based on characteristics of the data such as, for example, grouping all of the data gathered in a given week in a Block, in order to encourage efficient communications. Most data

[1]All numbers assume page sized operations, although the actual size of the page varies between the modules. We measured the XTend and Atmel energy numbers in prior work [25]. The ST energy numbers are from the datasheet [27] and assume a 1Mbps connection with the microcontroller. The Toshiba energy numbers are from [20].

reduction mechanisms will prove ineffective on tens of bytes of data and it is very difficult to transmit thousands of bytes of data in extremely unreliable networks.

Modules and Blocks are each given headers, which are central to DALi. The organization of these headers, as well as a view of how they are connected to the virtual and physical files, is shown in the pseudo-code on the right hand side of Figure 3. Module headers contain pointer information on where to find the Block headers as well as the overall Module size, the name of the corresponding virtual file, and information on which Blocks are empty or valid. These headers are stored in their own physical file so that they can be read and scanned independently of the data.

Block headers contain pointers to locate the sensor data in the physical sensor data file and the size of the Block, among other things. These headers are also stored in their own physical file; they are stored separately from the Module headers to create an easily scannable data hierarchy. The Module and Block headers hold the pointers to all of the other data on the node. When we refer to "opening" a Module or a Block in this work, we mean that the node is reading the header from Flash into RAM rather than reading the actual sensor data.

Additionally, at both the Module level and the Block level we store metadata summaries in order to assist with searches. Both Module metadata and Block metadata are stored in their own physical files. In Section 4, we will further discuss these summaries, as well as the versatile drill-down search structure that this data organization provides.

3.2 DALi: Module Naming Convention

Each Module name must be unique so that all data can be quickly identified in the network. We use a combination of the 2B (16-bit) node ID of the node generating the data[2], a 4B time stamp which counts seconds, and a 2B file counter that is simply incremented as virtual files are created and kept constant as Modules are added to the file. This setup is important when we attempt to move acknowledgements through the network, which we describe in Section 5.1.

A time stamp is not truly necessary. It could be replaced with a simple counter and DALi will still work properly. However, as we show in Section 4, time stamps allow for faster, more refined searches so we recommend that they be included in the implementation.

Finally, our decision of how to divide Modules into Blocks fits in well with this naming convention, since for communications purposes, a node can identify any data in the network at a Block granularity with just an 8B Module name and a 2B bit mask.

3.3 Module Name Location and the Time Ordered Structure

DALi requires data structures that can locate data quickly. One of our primary considerations is that nodes must process and store all incoming data, but that they will likely only encounter a small subset of the overall possibilities. Additionally, our resource constraints suggest that we use simple structures to minimize code size and RAM usage.

[2]For this work, we use a 2B node ID since the typical networks on which DALi will be deployed do not have more than 2^{16} nodes. However, this is easily changed if needed.

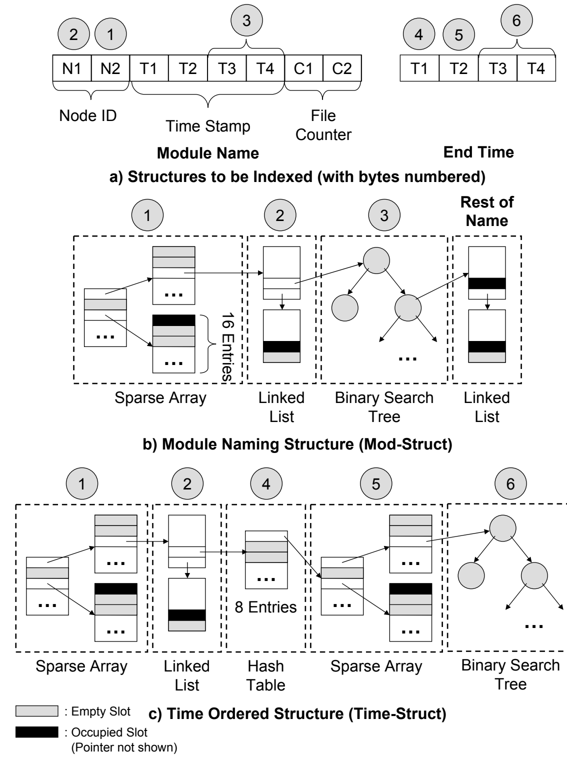

Figure 4: a) Parts of the Module name and the end time are used to index the simple data structures that link together to form b) the Module Naming Structure (Mod-Struct), used for general Module location, and c) the Time Ordered Structure (Time-Struct), used for temporal search. The linked lists in the structures are used to resolve collisions in the structures that precede them.

Although we want to abstract DALi from the physical storage medium, as we design these data structures it is also unwise to ignore the fundamental limitations of the Flash memory modules often found on sensor nodes. For example, Flash memory cannot overwrite data in a page unless the entire page is erased first so it is not possible to simply change pointers on the fly. Implementations using different physical storage media may benefit from different data structures than those discussed here, but the basic goals and principles of those implementations are ultimately the same.

3.3.1 *Module Naming Structure*

DALi includes naming structures to support both general name location as well as temporal search, which are depicted in Figure 4. The first component, the Module Naming Structure (Mod-Struct), is designed to keep search speed fast while minimizing the number of basic data structures. The top of the structure is a sparse array indexed by the LSB of the node's ID.

In DALi, a sparse array indexes a single byte of data. It breaks the byte into two 4-bit halves each used to index separate 16-entry tables as shown in Figure 5. The primary table indexes the 4 least significant bits and is created when the sparse array is created. It holds pointers to other tables which are indexed by the rest of the byte. When a slot in the primary table is used for the first time, the node creates the second 16-entry table. This structure enables us to index all

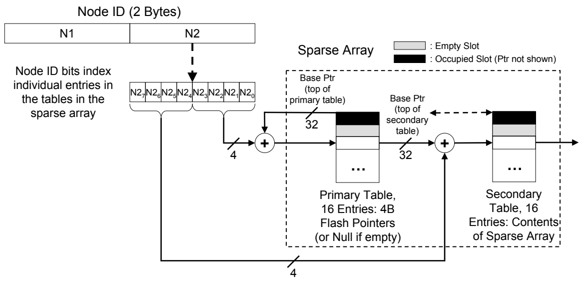

Figure 5: Example of how the LSB of one node ID is mapped to a sparse array in DALi. The primary table holds all 16 possible values of the 4 least significant bits of the node ID. Those entries each contain a pointer to a secondary table (or a Null value if no Node ID maps to that entry). The secondary table contains whatever data was to be stored in the sparse array.

possible one-byte entries with small tables that are easy to manage in Flash without worrying about index collisions.

If multiple node IDs in the Mod-Struct have the same LSB, we expand the sparse array with a linked list. Given the sparse nature of mobile and frequently disconnected networks, this list is likely to remain short.

That data structure points to a binary search tree indexed by the least significant 2B of the time stamp. As this number periodically wraps around, the systems we have explored have reasonably well-balanced trees. Nodes in the search tree each expand into a linked list in which entries hold the rest of the time stamp and the file counter.

Data structures for search typically use self-balancing trees, which tightly bound search times in all cases. However, the properties of Flash memory coupled with the node's memory constraints makes implementing self-balancing trees difficult. For our implementation, it is sufficient to use a combination of simple data structures and non-balancing trees. These trees either feature well-distributed indices that naturally yield a balanced tree or are small enough that they can degenerate into a linked list without a problem. However, more complicated structures have been developed for applications that must manage complex data structures in Flash [36, 37] and could be added to DALi in the future if necessary.

3.3.2 Time Ordered Structure

The first naming structure described so far is good for general search, but not for temporal search. As a result, as we implemented the Data Search services we found it necessary to add a Time Ordered Structure (Time-Struct), shown in Figure 4c. Rather than using the Module's start time to organize the tree, we use the time of the Module's last reading to emphasize searches starting with the most recent data.

The Time-Struct starts with a sparse array indexed by the node ID, just like the Mod-Struct. This array points to a hash table indexed by the three least significant bits of the most significant byte of the Module's end time. Given that our time stamp counts seconds, those three bits (bits 24-26 overall) can track more than four years of data. This table points to another sparse array, indexed by the second-most significant byte of the end time. Finally, this array points to a binary search tree indexed by the 2 LSBs of the time stamp; nodes in this tree also store the rest of the file name. However, unlike the Mod-Struct, the rest of the time stamp

	Item to Locate	Data Structure	Read From Flash
1	Bottom Half of LSB of Node ID	Sparse Array	Addr. of Second Half of Sparse Array
2	Top Half of LSB of Node ID	Sparse Array	Node ID, Addr. of Linked List (Step 3), Addr. of Hash Table (Step 4)
3	MSB of Node ID (Only if there is a collision on the LSB)	Linked List	Node ID, Addr. of Linked List (Step 3), Addr. of Hash Table (Step 4)
4	MSB of End Time	Hash Table	Addr. of First Half of Sparse Array
5	Bottom Half of 2nd MSB of End Time	Sparse Array	Addr. of Second Half of Sparse Array
6	Top Half of 2nd MSB of End Time	Sparse Array	Addr. of First Node in Binary Search Tree
7	2 LSBs of End Time	Binary Search Tree	Rest of the File Name, Addr. of Children in Tree, Addr of Module Header

Skip Step 3 if no collision on LSB of Node ID
Repeat as Needed (Step 3)
Repeat as Needed (Step 7)

Figure 6: A walk-through of a search on the Time-Struct. Step 3 is only executed in the event of an index collision in Step 2. The number of collisions is small, so it is a appropriate to resolve them with a hash table. Additionally, the Binary Search Tree in Step 7 will be small as well.

is indexed in the data structures above. A dataset storing 12B per minute never had more than two entries in a tree.

Figure 6 provides an example of the steps that DALi takes to execute searches in the Time-Struct. We evaluate both Time-Struct and the Mod-Struct further in Section 6.3.

3.3.3 Flash Memory Intricacies

Our assumptions focus on NOR flash memory and may not translate as well to NAND Flash modules. Since NAND memories will soon be used in sensor nodes, we briefly address ways to deal with them. With NAND Flash, a node cannot arbitrarily append data onto the end of a page without erasing that page first. To handle this, NAND Flash file systems often buffer one page worth of data (usually 512B) in RAM before writing to Flash. However, since one virtual file in DALi requires working with multiple physical files, this RAM buffering requirement would be prohibitive. We would handle this problem by writing the Module and Block headers and all of the metadata to a buffer in Flash and appending them to the appropriate physical files in page-sized groups. Additionally, erasures must be executed in multiple page chunks; however, Flash file systems typically fix this by allocating pages to files in groups large enough to erase.

3.4 The Underlying File System

One advantage of DALi, as shown in Figure 7, is that it resides above the file system. As the storage medium and low-level flash management details change, DALi can continue to take advantage of them.

Additionally, we note that DALi is more dependent on the speed of reads rather than the speed of writes to the virtual file. Reads will typically occur during data transmissions, when speed is at a premium for bandwidth and energy reasons, and due to the brevity of connections in these types of networks. However, virtual file writes can be buffered to Flash and performed off-line.

Each time the application writes sensor data to the virtual file, DALi has to write to multiple physical files. Likewise, the file system underneath DALi may have to perform multiple writes to non-volatile memory for each file write due to the nature of the storage medium. However, their performance and energy costs are more than outweighed by search and communications savings.

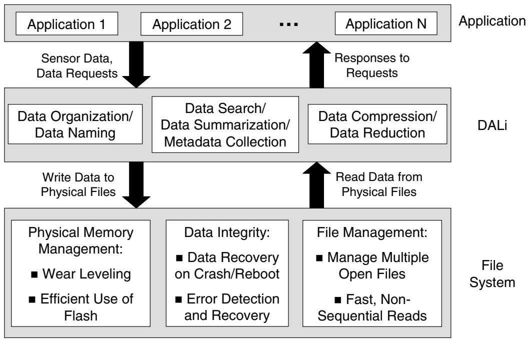

Figure 7: A summary of the services provided by the DALi layer and the services that DALi requires from the file system.

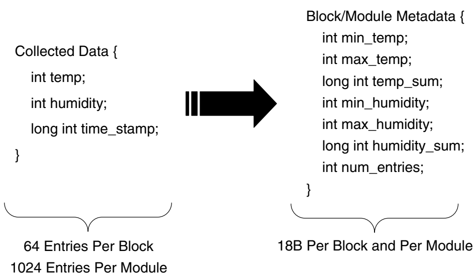

Figure 8: Example of the customizable metadata for a sensor collecting temperature and humidity readings. Metadata also includes start and end times for the data (not pictured).

4. DATA SEARCH: THE PERSISTENCE OF MEMORY

An effective Data Search mechanism for a sensor should be able to locate both specific data segments and sensed events stored on the node (generated by both itself and other nodes) and summarize them when necessary. As a result, DALi requires two drastically different types of searches: module name searches and data searches.

For both types of searches, our primary concern is that they execute quickly. This improves the node's response time to inquiries from other nodes, which in turn maximizes the effectiveness of brief encounters and minimizes idle radio energy consumption in longer ones.

Section 3.3 already discussed our methods for search-by-name. Here we discuss search-by-data-value. For example, search-by-data-value might be used to find all the stored temperature readings that are between 50 and 60 degrees. Such searches are important both internally to the node (e.g., since it may need to locate data for analysis or deletion) and externally to the network (e.g., for an application interested in only a subset of collected events).

DALi expedites data searches by storing customizable data summaries at both levels of the two-tiered data hierarchy, which creates a natural drill-down structure for the data. This metadata is generated during idle periods as Blocks and Modules are filled so that it is available on demand during communication periods.

Figure 8 shows a sample metadata structure for a sensor node collecting temperature and humidity readings. Given this structure, if we are looking for temperatures between 50 and 60 degrees, we need not drill down into any Modules that show a minimum above 60 or a maximum below 50 degrees. One can also use these summaries to estimate an average, or to detect outliers and interesting events. This metadata is customizable; the application developer just writes appropriate search algorithms to exploit it.

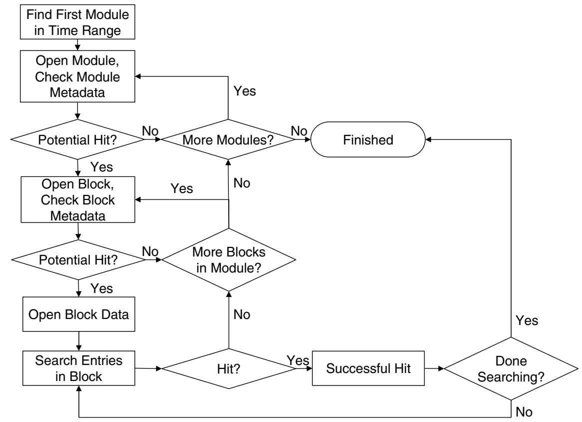

Figure 9: Flow chart of Data Sifting algorithm.

To accommodate time-based searches, we add start and end times to the Block metadata and end times to the Module metadata (the start time is already part of the file name). This preprocessing reduces the search size and in Section 6 we will show that it speeds up searches dramatically.

Data searches on sensor networks can typically be divided into two subsets, Data Sifting and Data Summarizing, which require slightly different search algorithms.

Data Sifting: Data Sifting, as depicted in Figure 9, involves locating specific entries or events. For spatial searches (e.g., find all datapoints within a bounding box of interest), one can use maximum and minimum metadata values to form bounding boxes, which the node can use to narrow the possible locations of a data hit. If the data point is in the module's bounding box, the abstraction layer drills down to the Block layer and looks at the metadata in each Block. If the point is in a Block's bounding box, then it searches the positional data itself. Hits are stored in a buffer provided by the application, and the search continues until it reaches a specified end-point (e.g., a maximum number of hits, the end of a time interval, etc.). This process works equally well when attempting to find ranges of non-spatial values too.

Data Summarizing: The other subset of searches, depicted in Figure 10, involve summarizing data. Here, rather than narrowing the volume of data to scan, the drill-down structure uses the metadata as a pregenerated data summary. It starts by comparing the start and end times of the first module in the time region specified. If the whole Module is within the time region, the abstraction layer just reads the Module metadata and moves onto the next Module. If not, however, it needs to drill down to the Block metadata and perform the same process at the Block level. If a Block is not entirely in the specified time region, the abstraction layer then drills down into the Block's actual sensor data.

Our summarizing algorithm ensures that the worst case summary involves scanning two Blocks of actual sensor data and 30 Block metadata structures (assuming that the time starts in the middle of the first Block of data in a Module and ends on the last Block of data in a separate Module) in addition to the Module metadata accesses. We will evaluate

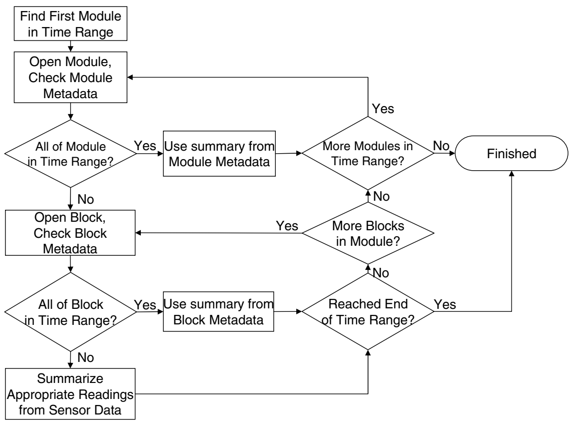

Figure 10: Flow chart of Data Summarizing algorithm.

both the Data Sifting and Data Summarizing algorithms in Section 6.4.

5. ADDITIONAL DALI SERVICES

This section examines first how DALi can save energy with its Data Naming mechanism by minimizing unnecessary transmissions, and then discusses how Data Reduction algorithms are integrated into the abstraction layer in order to make necessary transmissions more efficient.

5.1 Data Naming in Practice

Given the typical stream-oriented storage structure of sensor networks, the resource constraints of a typical sensor node, and the unreliable nature of communications in mobile sensor networks, it is very difficult for the node to know how much data has been successfully delivered to whom. Protocols often use opportunistic or epidemic communication approaches, which may replicate the data to reduce latency or just to improve the odds that the data successfully arrives at the sink. Once that data reaches the sink, acknowledgements should be propagated back through the network so that this data is not propagated further. Since individual sensed data items are small, we wish to acknowledge at a coarser quality. This is made possible by DALi's unique Module naming convention (see Section 3.2).

5.1.1 Case Study: Delete Lists

We can save energy by using the naming scheme just described to create acknowledgement streams we call delete lists. Delete lists are data structures that indicate that a particular segment of data has reached its destination, and, therefore, that the source and relay nodes can erase it. This concept was originally introduced in the first ZebraNet paper [14], and other works have theorized similar schemes [17], but to our knowledge no work until now has offered a practical or efficient implementation.

By significantly decreasing unnecessary transmissions, delete lists offer the potential of monumental energy savings. Such savings are magnified in unreliable networks, since the approach prunes retries if data was already successfully delivered via another path.

The biggest potential problem with this setup is that over time delete list entries could accumulate to the point that they themselves become difficult to transmit. However, as

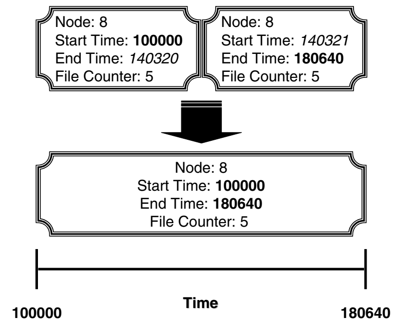

Figure 11: The "Melting Clocks": Coalescing delete list entries.

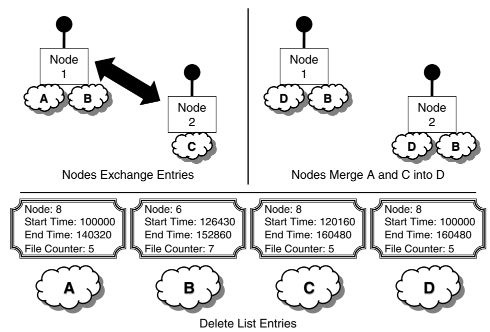

Figure 12: Nodes 1 and 2 exchange delete list entries related to nodes 6 and 8. The entries from node 8 (A and C) are from the same application and overlap in time, so they can be coalesced (to form entry D).

Figure 11 shows, DALi's Module naming convention provides a natural way to coalesce these entries into larger ranges as further data is successfully delivered.

Once all of the Blocks in a Module have been successfully delivered to the sink, the sink can generate a delete list message that specifies a node ID, a start time, an end time (determined from the enclosed data), and the file number. Both the sink and nodes in the network can coalesce these entries simply by adjusting the time range. Figure 12 shows an example of this process; since the entries for node 8 are part of the same file and overlap in time, they can be coalesced. This structure allows nodes to acknowledge multiple Modules of data with just a 12B packet payload, further improving the net energy savings, allowing nodes to store delete list entries indefinitely, and enabling nodes to flood entries through the network. This paper will evaluate delete lists in Section 6.5.

5.1.2 Data Naming: Network Services

To simplify communications, the data abstraction layer provides a service to the network layer that can negotiate a communication with a peer and ensure the efficient, reliable delivery of a Block. This process is shown in Figure 13. Such an organization offers out of order packet retransmissions even in severely memory constrained devices and makes it easy to ensure that complete Blocks of data are transmitted correctly over peer-to-peer links.

The naming mechanism of our data abstraction layer can also support communication operations in which nodes negotiate with each other about which packets to send. For example, the Module structure and naming conventions al-

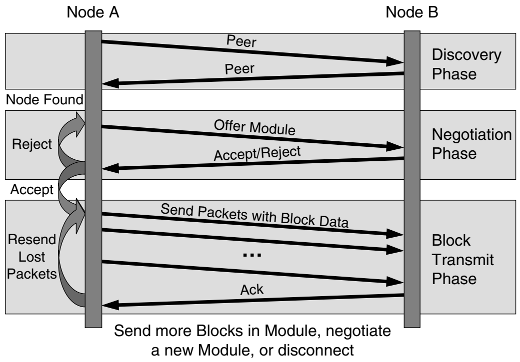

Figure 13: Sample network service for DALi that allows for out-of-order packet retransmissions.

low nodes to "offer" data to a neighbor, who can then turn it down if unneeded. This process is similar to the one proposed in SPIN [12, 15] in which a node advertises that it has data and other nodes in the area send requests when they would like to receive it.

5.2 Data Reduction

A sensor storage layer should support a range of data reduction functions. These include in-network computation, data aggregation, and compression. All of these aim to reduce energy consumption by computing locally in order to communicate less. We focus here on compression as an archetype for this style of data reduction.

The abstraction layer should make it easy to integrate compression into the system and, in most cases, the details of compression should be as well-hidden from the application as possible, since this simplifies application development. DALi also allows better control over when compression is performed. We prefer to compress data opportunistically during idle periods rather than doing it during a communication session which is likely to be busy and time-constrained.

Our implementation integrates the S-LZW with Mini-Cache compression algorithm. This is thoroughly evaluated in prior work [25], so we do not discuss it in detail here. However, there is one point that should be mentioned; although the node still compresses data in chunks of two Flash pages, in this paper we move to a newer, more energy efficient Flash module that organizes data in 256B pages (the module in the prior work used 264B pages). Our decision to use 512B Blocks in DALi is based on this new flash module and our results from the prior work[3].

5.2.1 System Integration

DALi expands upon the prior work in two key ways. First, since Flash writes are inexpensive and Flash memory is plentiful, we choose to keep *both* uncompressed and compressed versions of the data in Flash as opposed to discarding the uncompressed data once its compressed. Second, we optionally can decompress the data on intermediate relay nodes.

We keep the data in both its compressed form, so that its ready for transmission when connections become available, and its uncompressed form, so that the node can quickly answer queries. Once a Block is filled, its data is compressed and stored in a new Block which is tied to a Module that is independent of, but linked to, the Module with the uncompressed data. Both sets of data are stored in the physical file for sensor data. When compressed data is received, the node can rebuild the uncompressed stream and initiate a similar process.

An alternative possibility is to only keep data in its compressed form and uncompress it on demand. Our implementation did not employ this option because it would increase the worst case search times at a rate proportional to the number of Blocks that need to be decompressed. If the application rarely accesses the uncompressed data, however, this would be a reasonable solution.

Support for Data Reduction Beyond Compression: DALi's API can support other forms of simple on-node data reduction. The developer just needs to swap their function with the compression function. For these single-node reductions, the application developer can provide a data reduction "handler" function to be used instead of compression. For multi-node aggregations, which are more complex and less common, we expect the code to be a part of the application layer instead.

6. EVALUATION

This section evaluates the approaches discussed thus far, with a particular focus on DALi's Data Search algorithms. We first discuss our evaluation methodology. Then we examine the resource requirements of this implementation, evaluate our structures for search-by-name, and evaluate our Data Sifting and Data Summarizing algorithms using both traditional sensor data and spatial data from real world datasets. Finally, this section concludes by evaluating the potential energy benefits of delete lists.

6.1 Methodology

6.1.1 Platform

All experiments are conducted as real-system experiments on the ZebraNet v5.1 test board, which features a TI MSP430F1611 (10kB RAM, 48kB ROM) running at 4MHz [29] and an 8Mb ST Flash module [27]. This Flash module is smaller than the memories we would expect to use with DALi, but since DALi is not tied to any specific storage medium and we have the hardware available, it is a good starting point.

On this board the Flash communicates with the microcontroller at 1Mbps, but is capable of reads of up to 33Mbps. A faster microcontroller could decrease search times, but we use this one because it is among the more capable microcontrollers available for sensor nodes and we have it readily available on our hardware platform to run real evaluations.

The Flash module is broken into independent pages of 256B, and it takes 196μs to read a byte and 4.3ms to read a page ($\sim$16μs per byte with a 180μs overhead). These delays are directly reflected in our search times. As with most Flash modules, reads can be performed on data of any size.

To compare against communication times, we consider the XTend radio on the ZebraNet board. With this radio, we measured a baud rate of 7,394kbps (9,600kbps advertised), which translates to sending one 64B packet every 69.24ms.

To measure the execution times, we connect an oscilloscope to an unused pin on the microcontroller. We drive the

[3]Since the file system abstracts DALi from the Flash, the data size is more important than the number of pages. We use 512B Blocks rather than 528B Blocks in this work as more of a matter of convenience and consistency than as a requisite for functionality.

pin immediately before calling the appropriate algorithm, and release the pin immediately upon returning from the algorithm. This setup is accurate to within 10μs, which is appropriate given our 4MHz processor.

6.1.2 File System

Testing DALi requires an underlying file system. Since most of the available file systems are either based in TinyOS or designed for more capable processors, we built our own simple, stand-alone file system with ideas drawn from TinyOS-based file systems. Each file has an inode which contains pointers to index pages. Each index page contains a number of pointers, each of which leads to a page in Flash. This two layer structure allows us to locate any data in Flash in 2 independent reads of 2B each.

Sequential pages of data are linked together just like in ELF [6] so sequential data reads are easy to execute. Since pages are not allocated in order, if a file read crosses a page boundary, it incurs the 180μs Flash read overhead again; however, we guarantee that the 2B file indices never cross page boundaries.

6.1.3 Datasets

Our experiments use two real world datasets tested both individually and as two applications running simultaneously on the same node, each with their own virtual file of data. One dataset represents spatial data, or GPS positions, and the other represents traditional sensor data from simple low-energy sensors commonly found on sensor nodes.

For spatial data, we used a GPS trace of the Appalachian Trail [2]. When testing this dataset independently, we used the first 40,000 points as per-minute position readings. Each reading required 12B, 4B each for the latitude, longitude, and time stamp.

For traditional data, we used 18,000 entries from node 101 in the Great Duck Island (GDI) dataset [28]. For each entry, we stored all 11 sensor readings and replaced its time stamp with our own (accounting for one entry every five minutes as in their deployment) for a total of size 22B. We have filtered out duplicate entries and entries with errant sequence numbers. For our independent tests, we collect metadata based on the pressure and temperature data, and briefly evaluate the impact of collecting additional metadata.

6.2 System Resource Evaluation

Table 1 shows how our two datasets utilize the Flash when loaded into memory independently. Almost 90% of the memory is used to store the data in its compressed and uncompressed forms. The compressed data adds to the size of the Module headers, the Block headers, and the Module Naming Structure (Mod-Struct), but not to the metadata structures, since those are shared with the uncompressed data, nor to the Time Ordered Structure (Time-Struct), which only indexes the uncompressed data for search (see Section 6.3).

Each virtual file uses 5 physical files, plus the Mod-Struct and the Time-Struct are in their own files. Implementing DALi in this manner offers natural divisions at the software level. DALi could be set up to put all of these parts into one physical file, however, if limited by the underlying file system.

The primary RAM consumers in DALi are open Modules and open Blocks, which require 128B and 30B respectively.

	Appalachian Trail [2]		**Great Duck Island [28]**	
	Size(B)	Factor	Size(B)	Factor
Sensor Data	480,000	53.2%	396,000	51.8%
Compr. Data	324,031	35.9%	287,444	37.6%
Mod. Headers	13,440	1.5%	10,976	1.4%
Block Headers	41,932	4.6%	34,452	4.5%
Mod. Metadata	2,040	0.2%	1,666	0.2%
Block Metadata	36,214	4.0%	29,754	3.9%
Mod-Struct	2,504	0.3%	2,086	0.3%
Time-Struct	1,968	0.2%	1,896	0.2%
Totals				
Data	804,031	89.1%	683,444	89.4%
Overhead	98,098	10.9%	80830	10.6%

Table 1: Flash usage for our two experimental datasets.

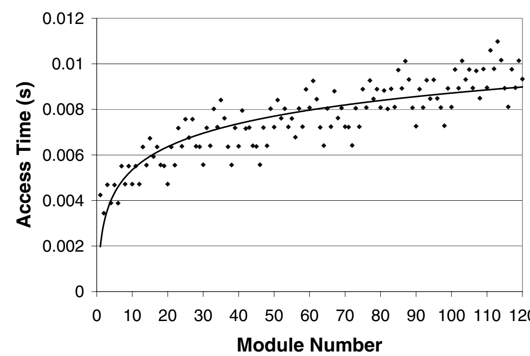

Figure 14: Time to find each of the Module headers in the Mod-Struct, shown with a logarithmic trendline. Module numbers 1 and 2 represent the compressed and uncompressed versions of the first Module of data.

However, our tests required no more than four open Modules and four open Blocks, and these numbers can likely be reduced further with occasional Flash buffering and refined coding in the Data Search algorithms.

The Mod-Struct and the Time-Struct are kept entirely in Flash, so they only require a file pointer in RAM. The S-LZW compression algorithm used in our implementation requires large buffers of 2kB and 512B, so when DALi needs buffer space (e.g., to scan a Block of data in RAM) it borrows those buffers from the compression algorithm [25].

6.3 Evaluating the Module Naming and Time Ordered Structures

To evaluate the Module Naming Structure (Section 3.3), we loaded the Appalachian dataset into a node's Flash. This dataset created 120 Modules, half for compressed data and half for uncompressed data. Then, we measured the time to find each of them in the Mod-Struct. The results are shown in Figure 14.

As can be expected with a binary search tree, access times increase logarithmically, but these times show that the tree is naturally balanced with this dataset. However, the access times vary fairly dramatically between the fastest access, which is less than 4ms, and the slowest access, which is around 11ms. This averages to 7.6ms, or just a little more than 10% of the time required to transmit a packet, across all 120 Modules.

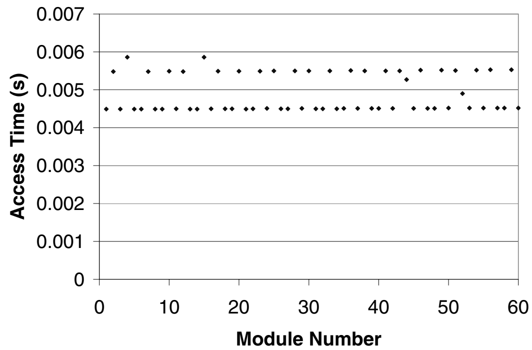

Figure 15: Time to find each of the Module headers in the Time-Struct. Only Module headers for uncompressed data are stored in this structure, and Module number 1 is the first Module of data.

Figure 15 shows the access times for the Time Ordered Structure (Time-Struct). Because this structure was intended to be used with the Data Search algorithms, we only add the Module headers for uncompressed data to the structure. The access times are faster on average and more consistent than those of the Mod-Struct, despite the fact that the Time-Struct has to access more underlying data structures to find the result. This is because the structure bounds the number of Flash accesses, unlike in the Mod-Struct with its constantly expanding tree. Two constant access times are apparent because the binary search tree at the end of the structure held up to two entries when loaded with this dataset. (The slight variations off of these times are caused by occasional reads across Flash page boundaries, which incur additional Flash read overhead).

The Time-Struct uses one more hash table and sparse array than the Mod-Struct, so its size grows faster. Even though the Time-Struct only held half the number of the entries as the Mod-Struct, it was 79% and 91% of the size for the Appalachian and GDI datasets respectively. However, its total size was still less than 0.3% of the amount of space required for the data itself for both datasets, so we do not consider it a problem.

For the search evaluations below, we use the Mod-Struct unless otherwise noted because it does not change our qualitative results for the datasets that we are using; however, given the results presented in the subsection, considerably larger datasets with many hundreds to thousands of Modules would benefit from a structure similar to the Time-Struct.

6.4 Data Search Evaluation

Next, we will evaluate the Data Sifting and Data Summarizing capabilities of DALi. Table 2 shows a few baseline measurements. Find, open, and read a Block of data represents the baseline search time. If the data we seek is in the first Module and Block, this is the time required to find the Module in Flash, open it, check its metadata, open the underlying Block, check its metadata, and read the sensor data into RAM. Once the data is in RAM, the node can scan it quickly.

If the data we seek is in a different Block in the open Module, there is a 2ms penalty to open the next Block and read its metadata. If the data is in a different *Module* (not just a different Block), there is at least an 8.6ms penalty to open the next Module. This varies based on the Mod-Struct access time, however.

Event	Execution Time
Find, open, and read a Block of data from scratch	19.25ms
Scan Block data in RAM	250μs
Open a Block and read a 12-18B metadata structure	$\sim$2ms
Open the next Module	8.6ms+

Table 2: Baseline access times with our datasets loaded into DALi.

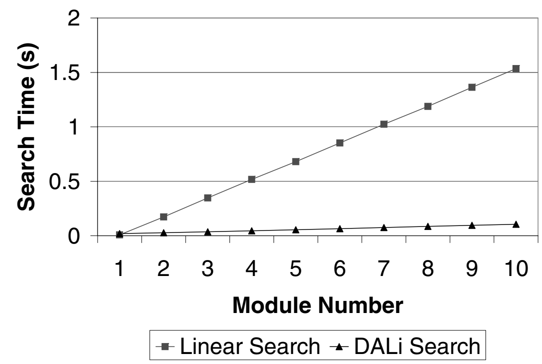

Figure 16: Comparison of a linear data search in Flash and a search using DALi. These searches look for the first data point in a Module, and Module number 1 is the first Module of data.

6.4.1 Data Sifting

To evaluate the Data Sifting mechanism, we use the Appalachian dataset and search for specific positions. The search algorithm can locate any position in a bounding box, but searching for a specific position gives us more control for testing purposes. The bounding boxes in the metadata help the node identify which Modules and Blocks may have the data. These algorithms also work for sifting other non-spatial data, since the metadata can hold any general maxima or minima.

DALi vs. linear search: Figure 16 compares the search times for finding the first entry in a Module with DALi versus searching for the equivalent position through a linear search of the Flash. For the linear search, we scan through all of the data as if it were stored in a large single-application circular buffer. While linear search requires 1.5s to scan just the first sixth of the data, DALi searches the same data in around 0.1s, already an order of magnitude improvement.

Searching over a time window: Although this algorithm is a significant improvement over linear search, scanning all 480,000B of data with this method still requires 740ms. To avoid such delays, we can use the Time-Struct to find the first Module in a given time period and begin our scan from that point. Figure 17 compares this method for finding the last data point in our dataset versus our original algorithm as we narrow the time range. The original algorithm has no fast way to find the starting point so it has to navigate over all of the Module headers. It does benefit a little from the narrower search ranges, though, since it is able to use the start times in the current and next Module names to eliminate Modules from the search without having to read the metadata from Flash.

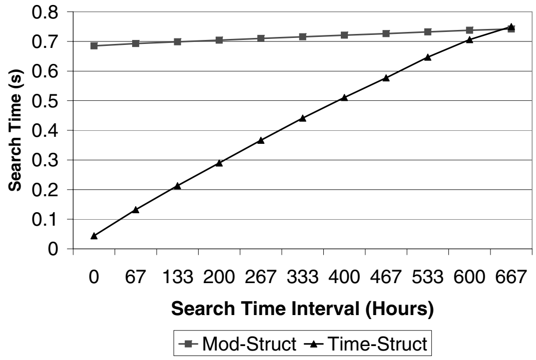

Figure 17: Comparison of a time restricted search in DALi with and without the Time-Struct on the Appalachian dataset, which assumes one reading per minute. Each search requests the last value in the time window.

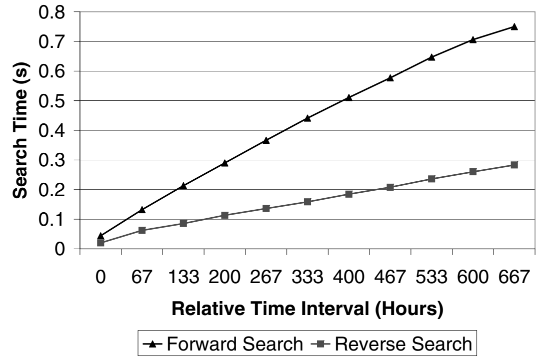

Figure 18: Comparison of two time restricted searches on the Appalachian dataset, which assumes one reading per minute. The reverse search starts at the end of the time window and moves backwards using direct pointers to Module headers, while the forward search starts at the beginning of the time window and moves forward by finding the address of the next Module in the Mod-Struct.

With the Time-Struct, however, the node can locate the correct starting point, which saves a great deal of time. For the last point in the graph, the relevant time range covers all of the data so the search times converge.

Searching the most recent data first: After the Time-Struct finds the appropriate starting point, DALi locates and scans through subsequent Module headers using the Mod-Struct (which requires 7.6ms per Module on average with this dataset). However, in many situations the user will be more interested in recent data than older data. Furthermore, in our implementation, the Module headers do not move once they are written to Flash. Therefore, as new Modules are created, DALi can place a pointer from the new header directly to the header for the previous Module. This allows the node to scan the newest Module first and walk backwards through preceding Modules without using the Mod-Struct. Although this simple improvement may not be appropriate when searching other node's data since Modules may be missing or arrive out of order, a node can ensure that it works when searching its own data; the Module headers are such a low percentage of the overall overhead (about 1.5%) that even if the underlying data is deleted, the headers can persist (missing data would just be skipped).

Figure 18 compares the two search structures based on the size of the time interval searched. The original algorithm starts at the oldest data and narrows the search until we hit the newest data. The reverse search algorithm does just the opposite. Although the searches are moving in opposite directions, this is a relevant comparison because the volume of data to be searched is the same. With this newer structure, search times grow at a far slower rate as the time interval increases, and even when the node has to search through all of the memory, it takes only about the same amount of time as is required to send four packets over the XTend radio.

6.4.2 Data Summarizing

With traditional queries, the node uses metadata both to provide bounding boxes for the data, like with spatial data, and to summarize data in advance to speed up summary queries. For the GDI dataset, the node extracted the maximum, minimum, sum, and average for the pressure and temperature readings from the pressure sensor.

Summarizing with DALi: Figure 19 shows the time required to summarize all of the temperature readings across a varying number of Blocks. It is described by:

$$t_{sum} \approx t_{search} + (N_{ModM} + N_{BlkM}) \times t_{meta} + N_B \times t_{scan} \quad (1)$$

where t_{search} is the time required to find the first Module in the time range to summarize, N_{ModM} and N_{BlkM} are the number of Module and Block metadata structures that must be read, t_{meta} is the time required to open the Modules and Blocks and read their metadata from Flash, N_B is the number of Blocks of data that need to be read and scanned, and t_{scan} is the time to read and scan them (although this particular experiment summarizes on Block boundaries so N_B is 0). The time required to finally compute the summary is far less than the Flash access time, so it is not included.

Figure 19 displays both the strengths and the weaknesses of our drill-down structure. To summarize the data in one Block, DALi needs to access the Module metadata to check the start and end times of the Module, open the first Block (i.e. read its header), and read its metadata. To summarize two Blocks, the node needs to open both Blocks and check each of their metadata. This trend continues through 15 Blocks. However, at 16 Blocks, there is a sudden drop in access time. This is because all of the data is already summarized in the Module metadata; there is no reason to drill down further. Therefore, the node actually summarizes the data faster than it did for just one Block. For 17 Blocks of data, the node accesses the metadata for the first Module and then moves to the second Module and its first Block.

Pregenerating additional metadata: Thus far, we have searched on granularities convenient to DALi, in which data is structured for communications rather than for human interests. Here we consider summarizing on one-day granularities, which are of interest to users and which do not map precisely to DALi Modules. The step-wise line in Figure 20 shows the amount of time required to summarize the GDI data in day granularities starting from the first reading. It takes a staircase form because of the Module/Block format. One day requires the node to read and merge the summaries for 12 full Blocks and half of a 13th block. Two days of data requires it to summarize 1 full Module and 10 additional Blocks, which takes about the

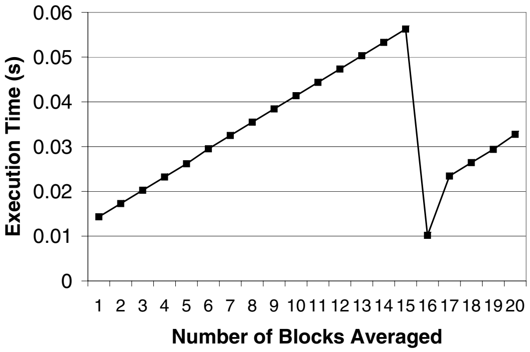

Figure 19: Time to summarize the data in a given number of Blocks.

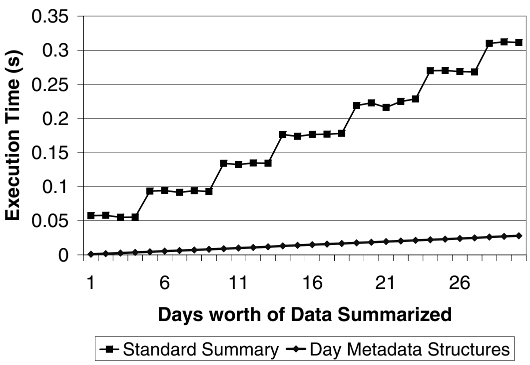

Figure 20: Time to summarize the data on day-long granularities with the basic search and with tailored metadata.

same amount of time. Between the fourth and fifth points, though, there is a jump because four days requires the node to summarize just 3 Modules and 3 Blocks, while five days requires it to summarize 3 Modules and 15 Blocks.

DALi's structure is flexible and makes few assumptions as to what data will be present; however, if a node knows that it has all of a node's data over a particular time period (as it does for its own data), it can create a new physical file and generate and store metadata on that data at a granularity meaningful to the end user. When the node receives a query for a data summary on this granularity, the node can simply read the answers from the file. For this purpose, we generated a file of metadata on day granularities and reran the test. This results in much faster execution, as shown in the lower line in Figure 20.

How does Metadata Size Impact Search Times? For the GDI data, we only collected metadata for a subset of the readings. However, a different application may want additional metadata. To evaluate the impact of larger metadata structures, we added maxima and minima for the other nine sensor readings recorded in the GDI deployment, increasing the metadata from 18B to 52B. This increases search times only slightly (less than 0.5ms per metadata structure scanned) because it takes longer to read it from Flash, so we do not consider it to be an issue with this implementation. However, in a different implementation in which the metadata needs to be transmitted to allow other nodes to search the data, there would be an energy penalty associated with larger metadata structures.

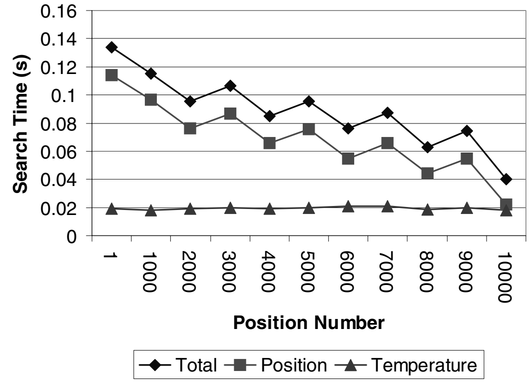

Figure 21: Time to find the last known temperature at a given location.

6.4.3 *Spatially-Aware Queries*

A more complicated, but practical subspace of the search domain is spatially-aware queries. A sensor node would likely control the GPS and the general sensors through separate applications. Under DALi, each application has its own virtual file. For our experiments, we loaded the first 10,000 readings for both the Appalachian and GDI datasets onto the node (using one application for each dataset), and the Appalachian dataset was adjusted to assume that one reading was recorded every five minutes like in the GDI deployment. Although the GDI deployment was not mobile, the temperature readings work well for these experiments.

Searching for temperature based on position: The first experiment asks a node for the most recent temperature and time stamp in a given geographic region. It performs a location-based search and then retrieves the temperature based on a time stamp. We performed the search for 11 points spaced out through the 10,000 loaded into memory. The results are shown in Figure 21 and are broken down by application. In the first step, the node retrieves the time stamp for the appropriate GPS position. The sawtooth shape in the access times for the GPS positions is caused by the Block metadata organization as shown in Figure 19. The second step returns the temperature based on the time stamp, which only requires finding and reading the correct Block, so the search time is almost constant. In all cases, finding the temperature took less time than the 138.5ms required to send two packets over the radio.

Searching for positions with a given temperature: The second experiment requests all of the locations and times when the temperature was in a given range. This search is more complex than the previous one, because there are multiple answers. Figure 22 shows the times required to find each of the first 42 answers (which fills one Block of data) for a sample temperature range.

Most of the time delays are negligible, and this is because hits often come in closely grouped batches and the Block data has already been read into RAM for those points. The first peak is the overhead of finding the first hit, and subsequent peaks represent the time required to move between batches of hits. This time is correlated to the amount of Block and Module metadata that the node must skim to find the next hit. The positional times are typically shorter because more positional data fits into a Block than GDI data, so there are fewer Block accesses.

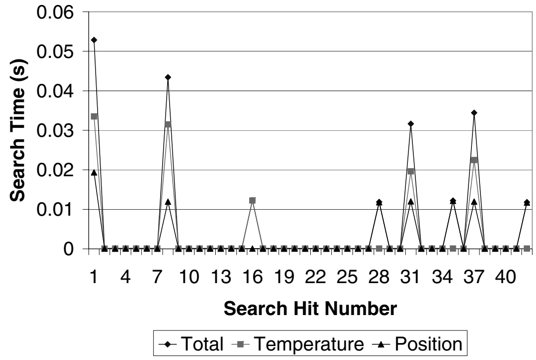

Figure 22: Time to find each subsequent hit when searching for all positions and times in which the temperature was in a given range.

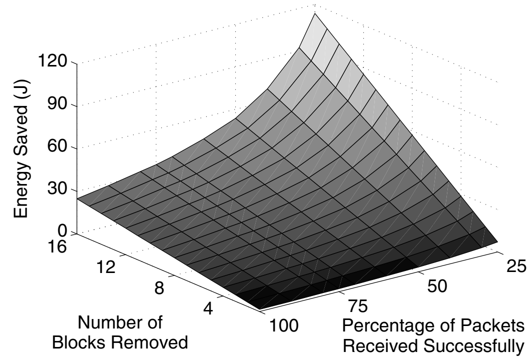

Figure 23: Energy saved by sending a delete list packet rather than sending the sensor data over the XTend Radio.

Through this search, a node could opt to send the hits as is or to coalesce individual groups of entries into one entry with a start time, an end time, and a bounding box; the later of these options saves space if more than two entries can be merged and if the end user is content with receiving a bounding box rather than more specific positional data.

Benefit of using separate applications to collect spatial and traditional data instead of just one: In these experiments, DALi knew the number of readings per Block and the frequency of readings. In time-based searches, once the system found the first Module in the search through the Time-Struct, this information allowed it to correctly calculate which Block to search first. However, if this data structure were not present, these values would vary from Block to Block (as would likely be the case if data from multiple applications was stored in a single virtual file). This increases search times because DALi needs to sequentially read the metadata for each Block in the Module until it finds the correct starting point. Additionally, it must determine the size of each reading individually before processing it, but this operation is RAM based so the resulting delay is much less significant.

6.5 Delete Lists Evaluation

This subsection presents a brief evaluation of the delete lists presented in Section 5.1, which can reduce energy consumption by multiple orders of magnitude. Assuming that we send exactly one 12B delete list entry per packet, Figure 23 shows the potential energy savings from the XTend radio as the number of 512B Blocks covered by the entry increases from 1 to 16 and as the successful packet reception rate drops from 100% to 25%.

This graph is based on the power model of the XTend radio presented in prior work [25], which is derived from transmit and receive energy numbers measured on our sensor platform. Even accounting for radio's high activation overhead, we save close to 10X energy by sending a delete list packet over a single Block of data and a maximum of a factor of 170X versus sending a full Module in an unreliable network using the network services described in Section 5.1.2. The potential savings increase as we represent more and more Modules in a single coalesced delete list entry.

These results only vary based on the volume of data collected. Looking at the Appalachian dataset, for example, we can qualitatively compare delete lists in DALi with a theoretical example of what they would look like if data were arbitrarily stored in Flash. Given 12B readings which each include a 4B time stamp, DALi can store 42 per Block and 672 per Module. Without DALi, each reading would have to be acknowledged individually, probably by the time stamp, so we assume 4B per delete list entry. Therefore, to offer a delete list that represents 672 readings, it would require 2688B before considering packet overheads. This places a tremendous strain on unreliable networks. DALi, on the other hand, requires just 12B to represent the same volume of information.

Without DALi, once a node starts receiving information from other nodes it cannot easily identify consecutive delete list entries; therefore, it cannot coalesce these entries and the corresponding readings must be found and marked for deletion individually in Flash. DALi, however, can use its search functionalities to locate the data and mark it for deletion in Block sized chunks which is far simpler and faster.

7. CONCLUSION

This paper explores DALi, a communication-centric Data Abstraction Layer that reorganizes data from the viewpoint of sparse mobile and frequently disconnected sensor nodes in order to provide the system with services for three important system services: Data Search, Naming, and Reduction.

DALi's natural drill-down mechanism can quickly and efficiently locate specific data, summarize chunks of data, and perform more complicated but practical searches over data collected by multiple applications. Simple data structures and algorithmic improvements dramatically decrease search times, which conserves bandwidth during communications periods. This helps to maximize the effectiveness of brief encounters and minimize radio on-time during longer ones.

Through our Data Naming scheme, nodes can easily identify data in memory and use manageable delete lists entries to remove data from the network that has already been delivered to the sink. These mechanisms prevent unnecessary communications and, in the process, conserve bandwidth and reduce energy consumption, potentially by multiple orders of magnitude.

DALi's overall structure is designed to make communications simpler and more efficient. This structure allows the node to generate the naming and metadata structures on which all of our services are based. With sensor networks increasingly moving towards sparse, mobile networks of opportunistically-communicating nodes, DALi offers the needed infrastructure to support efficient data queries and storage for these important but challenging systems.

Acknowledgements

We thank Kevin Wayne and James Donald for their advice on implementing data structures on Flash memories; Pei Zhang for helping to set up the sensor hardware and measurements; and our shepherd for this paper, William Griswold (UCSD). This work was supported in part by NSF ITR program (CCR 0205214).

8. REFERENCES

[1] Aleph One. Yet Another Flash File System. `http://www.aleph1.co.uk/yaffs`.

[2] Appalachian Trail Conservancy. Appalachian Trail GIS and GPS Data. `http://www.appalachiantrail.org/`, Mar. 2002.

[3] ATMEL. AT45DB041B, 4M bit, 2.7-Volt Only Serial-Interface Flash with Two 264-Byte SRAM Buffers data sheet. `http://www.atmel.com/`, June 2003.

[4] D. Carlson. Sensor Networks in Polar Regions: Urgent Needs, Difficult Challenges. Keynote Presentation, ACM Conf. on Embedded Networked Sensor Systems (SenSys), Nov. 2006.

[5] B. Cohen. Incentives Build Robustness in BitTorrent. `http://www.bittorrent.org/bittorrentecon.pdf`, May 2003.

[6] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

[7] C. Decker, M. Beigl, and A. Krohn. A File System for System Programming in Ubiquitous Computing. In *Intl. Conf. on the Architecture of Computing Systems*, Mar. 2005.

[8] K. Fall. A Delay Tolerant Network Architecture for Challenged Internets. In *Proc. of the Special Interest Group on Data Communications Conf. (SIGCOMM)*, 2003.

[9] D. Ganesan, B. Greenstein, D. Estrin, et al. Multi-resolution Storage and Search in Sensor Networks. *ACM Transactions on Storage*, 1(3):277–315, Aug. 2005.

[10] D. Gay. Design of Matchbox, the Simple Filing System for Motes. `http://www.tinyos.net`, Aug. 2003.

[11] J. Gehrke and S. Madden. Query Processing in Sensor Networks. *Pervasive Computing*, Jan. 2004.

[12] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. *Wireless Networks*, 8:169–185, 2002.

[13] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Intl. Conf. on Mobile Computing and Networking (MOBICOM)*, Aug. 2000.

[14] P. Juang, H. Oki, Y. Wang, et al. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[15] J. Kulik, W. Heinzelman, and H. Balakrishnan. Negotiation-Based Protocols for Disseminating Information in Wireless Sensor Networks. *Wireless Networks*, 8:169–185, 2002.

[16] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, et al. Efficient Indexing Data Structures for Flash-Based Sensor Devices. In *ACM Transactions on Storage*, 2006.

[17] A. Lindgren, A. Doria, and O. Schelen. Probabilistic Routing in Intermittently Connected Networks. In *Intl. Wksp. on Service Assurance with Partial and Intermittent Resources*, Sept. 2004.

[18] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[19] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *SIGMOD*, 2003.

[20] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2006.

[21] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Ultra-Low Power Data Storage for Sensor Networks. In *Proc. of the ACM Conf. on Information Processing in Sensor Networks (IPSN-SPOTS)*, Apr. 2006.

[22] Maxstream, Inc. XTend OEM RF Module: Product Manual v1.2.4. `http://www.maxstream.net/`, Oct. 2005.

[23] S. Rajagopalan and C.-C. Shen. A Cross-Layer, Decentralized BitTorrent for Mobile Ad hoc Networks. In *Proc. of MOBIQUITOUS'06*, July 2006.

[24] S. Ratnasamy, B. Karp, S. Shenker, et al. Data-Centric Storage in Sensornets with GHT, a Geographic Hash Table. In *Mobile Networks and Applications (MONET), Journal of Special Issues on Mobility of Systems, Users, Data, and Computing: Special Issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks*, 2003.

[25] C. M. Sadler and M. Martonosi. Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2006.

[26] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: Modeling a Three-tier Architecture for Sparse Sensor Networks. In *Proc. of IEEE Workshop on Sensor Network Protocols and Applications*, May 2003.

[27] ST. M45PE40: 8 Mbit, Low Voltage, Page-Erasable Serial Flash Memory With Byte-Alterability and a 33 MHz SPI Bus Interface. `http://www.st.com/`, Feb. 2006.

[28] R. Szewczyk, A. Mainwaring, J. Polastre, et al. An Analysis of a Large Scale Habitat Monitoring Application. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

[29] Texas Instruments. MSP430x161x Mixed Signal Microcontroller Datasheet. `http://www.ti.com/`, Mar. 2005.

[30] S. Tilak, B. Pisupati, K. Chiu, et al. A File System Abstraction for Sense and Respond Systems. In *Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services*, 2005.

[31] Toshiba America Electronic Components, Inc. (TAEC). Datasheet: TC58DVG02A1FT00. `http://www.toshiba.com/taec`, Jan. 2003.

[32] A. Vahdat and D. Becker. Epidemic routing for partially connected ad hoc networks. In *Technical Report CS-200006, Duke University*, Apr. 2000.

[33] P. S. Vellore, P. Gillard, and R. Venkatesen. MBEAN: multicasting in BitTorrent enabled ad hoc networks. In *Proc. of the 2005 Intl. Conf. on Wireless Networks, Communications and Mobile Computing*, 2005.

[34] Y. Wang, S. Jain, M. Martonosi, and K. Fall. Erasure-Based Routing for Opportunistic Networks. In *Proc. of the Special Interest Group on Data Communications Conf. (SIGCOMM)*, Aug. 2005.

[35] D. Woodhouse. Journalling Flash File System. `http://sources.redhat.com/jffs2/jffs2.pdf`.

[36] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An Efficient B-Tree Layer for Flash-Memory Storage Systems. In *Intl. Conf. on Real-Time and Embedded Computing Systems and Applications*, 2003.

[37] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An Efficient R-Tree Implementation over Flash-Memory Storage Systems. In *Intl. Symp. on Advances in Geographic Information Systems*, Nov. 2003.

[38] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record*, 31(3), Sept. 2002.

[39] P. Zhang, C. M. Sadler, S. A. Lyon, and M. Martonosi. Hardware Design Experiences in ZebraNet. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.