TSAR: A Two Tier Sensor Storage Architecture Using Interval Skip Graphs*

Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
pjd@cs.umass.edu, dganesan@cs.umass.edu, shenoy@cs.umass.edu

ABSTRACT

Archival storage of sensor data is necessary for applications that query, mine, and analyze such data for interesting features and trends. We argue that existing storage systems are designed primarily for flat hierarchies of homogeneous sensor nodes and do not fully exploit the multi-tier nature of emerging sensor networks, where an application can comprise tens of tethered proxies, each managing tens to hundreds of untethered sensors. We present TSAR, *a fundamentally different storage architecture that envisions separation of data from metadata by employing local archiving at the sensors and distributed indexing at the proxies. At the proxy tier, TSAR employs a novel multi-resolution ordered distributed index structure, the Interval Skip Graph, for efficiently supporting spatio-temporal and value queries. At the sensor tier, TSAR supports energy-aware adaptive summarization that can trade off the cost of transmitting metadata to the proxies against the overhead of false hits resulting from querying a coarse-grain index. We implement TSAR in a two-tier sensor testbed comprising Stargate-based proxies and Mote-based sensors. Our experiments demonstrate the benefits and feasibility of using our energy-efficient storage architecture in multi-tier sensor networks.*

Categories and Subject Descriptors: C.2.4 [Computer – Communication Networks]: Distributed Systems

General Terms: Algorithms, performance, experimentation.

Keywords: Wireless sensor networks, archival storage, indexing methods.

1. Introduction

1.1 Motivation

Many different kinds of networked data-centric sensor applications have emerged in recent years. Sensors in these applications sense the environment and generate data that must be processed, filtered, interpreted, and archived in order to provide a useful infrastructure to its users. To achieve its goals, a typical sensor application needs access to both live and past sensor data. Whereas access to live data is necessary in monitoring and surveillance applications, access to past data is necessary for applications such as mining of sensor logs to detect unusual patterns, analysis of historical trends, and post-mortem analysis of particular events. Archival storage of past sensor data requires a storage system, the key attributes of which are: where the data is stored, whether it is indexed, and how the application can access this data in an energy-efficient manner with low latency.

There have been a spectrum of approaches for constructing sensor storage systems. In the simplest, sensors stream data or events to a server for long-term archival storage [3], where the server often indexes the data to permit efficient access at a later time. Since sensors may be several hops from the nearest base station, network costs are incurred; however, once data is indexed and archived, subsequent data accesses can be handled locally at the server without incurring network overhead. In this approach, the storage is centralized, reads are efficient and cheap, while writes are expensive. Further, all data is propagated to the server, regardless of whether it is ever used by the application.

An alternate approach is to have each sensor store data or events locally (e.g., in flash memory), so that all writes are local and incur no communication overheads. A read request, such as whether an event was detected by a particular sensor, requires a message to be sent to the sensor for processing. More complex read requests are handled by flooding. For instance, determining if an intruder was detected over a particular time interval requires the request to be flooded to all sensors in the system. Thus, in this approach, the storage is distributed, writes are local and inexpensive, while reads incur significant network overheads. Requests that require flooding, due to the lack of an index, are expensive and may waste precious sensor resources, even if no matching data is stored at those sensors. Research efforts such as Directed Diffusion [17] have attempted to reduce these read costs, however, by intelligent message routing.

Between these two extremes lie a number of other sensor storage systems with different trade-offs, summarized in Table 1. The geographic hash table (GHT) approach [24, 26] advocates the use of an in-network index to augment the fully distributed nature of sensor storage. In this approach, each data item has a key associated with it, and a distributed or geographic hash table is used to map keys to nodes that store the corresponding data items. Thus, writes cause data items to be sent to the hashed nodes and also trigger updates to the in-network hash table. A read request requires a lookup in the in-network hash table to locate the node that stores the data

*This work was supported in part by National Science Foundation grants EEC-0313747, CNS-0325868 and EIA-0098060.

SenSys'05, November 2–4, 2005, San Diego, California, USA.

item; observe that the presence of an index eliminates the need for flooding in this approach.

Most of these approaches assume a flat, homogeneous architecture in which every sensor node is energy-constrained. In this paper, we propose a novel storage architecture called *TSAR*[1] that reflects and exploits the multi-tier nature of emerging sensor networks, where the application is comprised of tens of tethered sensor proxies (or more), each controlling tens or hundreds of untethered sensors. TSAR is a component of our PRESTO [8] predictive storage architecture, which combines archival storage with caching and prediction. We believe that a fundamentally different storage architecture is necessary to address the multi-tier nature of future sensor networks. Specifically, the storage architecture needs to exploit the resource-rich nature of proxies, while respecting resource constraints at the remote sensors. No existing sensor storage architecture explicitly addresses this dichotomy in the resource capabilities of different tiers.

Any sensor storage system should also carefully exploit current technology trends, which indicate that the capacities of flash memories continue to rise as per Moore's Law, while their costs continue to plummet. Thus it will soon be feasible to equip each sensor with 1 GB of flash storage for a few tens of dollars. An even more compelling argument is the energy cost of flash storage, which can be as much as two orders of magnitude lower than that for communication. Newer NAND flash memories offer very low write and erase energy costs — our comparison of a 1GB Samsung NAND flash storage [16] and the Chipcon CC2420 802.15.4 wireless radio [4] in Section 6.2 indicates a 1:100 ratio in per-byte energy cost between the two devices, even before accounting for network protocol overheads. These trends, together with the energy-constrained nature of untethered sensors, indicate that local storage offers a viable, energy-efficient alternative to communication in sensor networks.

TSAR exploits these trends by storing data or events locally on the energy-efficient flash storage at each sensor. Sensors send concise identifying information, which we term *metadata*, to a nearby proxy; depending on the representation used, this metadata may be an order of magnitude or more smaller than the data itself, imposing much lower communication costs. The resource-rich proxies interact with one another to construct a distributed index of the metadata reported from all sensors, and thus an index of the associated data stored at the sensors. This index provides a unified, logical view of the distributed data, and enables an application to query and read past data efficiently — the index is used to pinpoint all data that match a read request, followed by messages to retrieve that data from the corresponding sensors. In-network index lookups are eliminated, reducing network overheads for read requests. This separation of data, which is stored at the sensors, and the metadata, which is stored at the proxies, enables TSAR to reduce energy overheads at the sensors, by leveraging resources at tethered proxies.

1.2 Contributions

This paper presents TSAR, a novel two-tier storage architecture for sensor networks. To the best of our knowledge, this is the first sensor storage system that is explicitly tailored for emerging multi-tier sensor networks. Our design and implementation of TSAR has resulted in four contributions.

At the core of the TSAR architecture is a novel distributed index structure based on interval skip graphs that we introduce in this paper. This index structure can store coarse summaries of sensor data and organize them in an ordered manner to be easily searchable. This data structure has $O(\log n)$ expected search and update complexity. Further, the index provides a logically unified view of all data in the system.

Second, at the sensor level, each sensor maintains a local archive that stores data on flash memory. Our storage architecture is fully stateless at each sensor from the perspective of the metadata index; all index structures are maintained at the resource-rich proxies, and only direct requests or simple queries on explicitly identified storage locations are sent to the sensors. Storage at the remote sensor is in effect treated as appendage of the proxy, resulting in low implementation complexity, which makes it ideal for small, resource-constrained sensor platforms. Further, the local store is optimized for time-series access to archived data, as is typical in many applications. Each sensor periodically sends a summary of its data to a proxy. TSAR employs a novel adaptive summarization technique that adapts the granularity of the data reported in each summary to the ratio of false hits for application queries. More fine grain summaries are sent whenever more false positives are observed, thereby balancing the energy cost of metadata updates and false positives.

Third, we have implemented a prototype of TSAR on a multi-tier testbed comprising Stargate-based proxies and Mote-based sensors. Our implementation supports spatio-temporal, value, and range-based queries on sensor data.

Fourth, we conduct a detailed experimental evaluation of TSAR using a combination of EmStar/EmTOS [10] and our prototype. While our EmStar/EmTOS experiments focus on the scalability of TSAR in larger settings, our prototype evaluation involves latency and energy measurements in a real setting. Our results demonstrate the logarithmic scaling property of the sparse skip graph and the low latency of end-to-end queries in a duty-cycled multi-hop network .

The remainder of this paper is structured as follows. Section 2 presents key design issues that guide our work. Section 3 and 4 present the proxy-level index and the local archive and summarization at a sensor, respectively. Section 5 discusses our prototype implementation, and Section 6 presents our experimental results. We present related work in Section 7 and our conclusions in Section 8.

2. Design Considerations

In this section, we first describe the various components of a multi-tier sensor network assumed in our work. We then present a description of the expected usage models for this system, followed by several principles addressing these factors which guide the design of our storage system.

2.1 System Model

We envision a multi-tier sensor network comprising multiple tiers — a bottom tier of untethered remote sensor nodes, a middle tier of tethered sensor proxies, and an upper tier of applications and user terminals (see Figure 1).

The lowest tier is assumed to form a dense deployment of low-power sensors. A canonical sensor node at this tier is equipped with low-power sensors, a micro-controller, and a radio as well as a significant amount of flash memory (e.g., 1GB). The common constraint for this tier is energy, and the need for a long lifetime in spite of a finite energy constraint. The use of radio, processor, RAM, and the flash memory all consume energy, which needs to be limited. In general, we assume radio communication to be substantially more expensive than accesses to flash memory.

The middle tier consists of power-rich sensor proxies that have significant computation, memory and storage resources and can use

[1]TSAR: Tiered Storage ARchitecture for sensor networks.

Table 1: Characteristics of sensor storage systems

System	Data	Index	Reads	Writes	Order preserving
Centralized store	Centralized	Centralized index	Handled at store	Send to store	Yes
Local sensor store	Fully distributed	No index	Flooding, diffusion	Local	No
GHT/DCS [24]	Fully distributed	In-network index	Hash to node	Send to hashed node	No
TSAR/PRESTO	Fully distributed	Distributed index at proxies	Proxy lookup + sensor query	Local plus index update	Yes

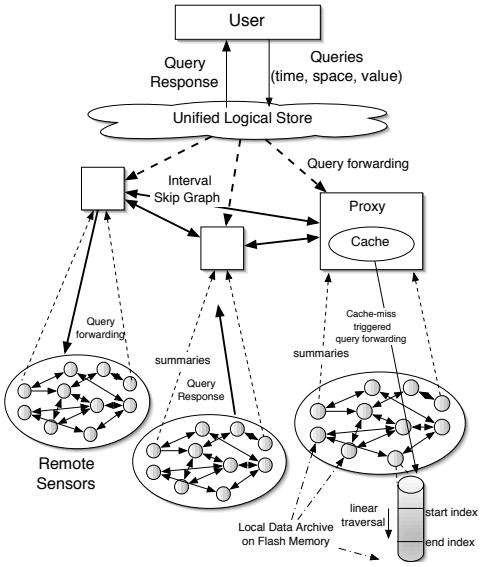


Figure 1: Architecture of a multi-tier sensor network.

these resources continuously. In urban environments, the proxy tier would comprise a tethered base-station class nodes (e.g., Crossbow Stargate), each with with multiple radios—an 802.11 radio that connects it to a wireless mesh network and a low-power radio (e.g. 802.15.4) that connects it to the sensor nodes. In remote sensing applications [10], this tier could comprise a similar Stargate node with a solar power cell. Each proxy is assumed to manage several tens to hundreds of lower-tier sensors in its vicinity. A typical sensor network deployment will contain multiple geographically distributed proxies. For instance, in a building monitoring application, one sensor proxy might be placed per floor or hallway to monitor temperature, heat and light sensors in their vicinity.

At the highest tier of our infrastructure are applications that query the sensor network through a query interface[20]. In this work, we focus on applications that require access to past sensor data. To support such queries, the system needs to archive data on a persistent store. Our goal is to design a storage system that exploits the relative abundance of resources at proxies to mask the scarcity of resources at the sensors.

2.2 Usage Models

The design of a storage system such as TSAR is affected by the queries that are likely to be posed to it. A large fraction of queries on sensor data can be expected to be spatio-temporal in nature. Sensors provide information about the physical world; two key attributes of this information are *when* a particular event or activity occurred and *where* it occurred. Some instances of such queries include the time and location of target or intruder detections (e.g., security and monitoring applications), notifications of specific types of events such as pressure and humidity values exceeding a threshold (e.g., industrial applications), or simple data collection queries which request data from a particular time or location (e.g., weather or environment monitoring).

Expected queries of such data include those requesting ranges of one or more attributes; for instance, a query for all image data from cameras within a specified geographic area for a certain period of time. In addition, it is often desirable to support efficient access to data in a way that maintains spatial and temporal ordering. There are several ways of supporting range queries, such as locality-preserving hashes such as are used in DIMS [18]. However, the most straightforward mechanism, and one which naturally provides efficient ordered access, is via the use of *order-preserving* data structures. Order-preserving structures such as the well-known B-Tree maintain relationships between indexed values and thus allow natural access to ranges, as well as predecessor and successor operations on their key values.

Applications may also pose *value-based* queries that involve determining if a value v was observed at any sensor; the query returns a list of sensors and the times at which they observed this value. Variants of value queries involve restricting the query to a geographical region, or specifying a range (v_1, v_2) rather than a single value v. Value queries can be handled by indexing on the values reported in the summaries. Specifically, if a sensor reports a numerical value, then the index is constructed on these values. A search involves finding matching values that are either contained in the search range (v_1, v_2) or match the search value v exactly.

Hybrid value and spatio-temporal queries are also possible. Such queries specify a time interval, a value range and a spatial region and request all records that match these attributes – "find all instances where the temperature exceeded 100^o F at location R during the month of August". These queries require an index on both time and value.

In TSAR our focus is on range queries on value or time, with planned extensions to include spatial scoping.

2.3 Design Principles

Our design of a sensor storage system for multi-tier networks is based on the following set of principles, which address the issues arising from the system and usage models above.

- **Principle 1:** *Store locally, access globally:* Current technology allows local storage to be significantly more energy-efficient than network communication, while technology trends show no signs of erasing this gap in the near future. For maximum network life a sensor storage system should leverage the flash memory on sensors to archive data locally, substituting cheap memory operations for expensive radio transmission. But without efficient mechanisms for retrieval, the energy gains of local storage may be outweighed by communication costs incurred by the application in searching for data. We believe that if the data storage system provides the abstraction of a single logical store to applications, as

does TSAR, then it will have additional flexibility to optimize communication and storage costs.

- **Principle 2:** *Distinguish data from metadata:* Data must be identified so that it may be retrieved by the application without exhaustive search. To do this, we associate *metadata* with each data record — data fields of known syntax which serve as identifiers and may be queried by the storage system. Examples of this metadata are data attributes such as location and time, or selected or summarized data values. We leverage the presence of resource-rich proxies to index metadata for resource-constrained sensors. The proxies share this metadata index to provide a unified logical view of all data in the system, thereby enabling efficient, low-latency lookups. Such a tier-specific separation of data storage from metadata indexing enables the system to exploit the idiosyncrasies of multi-tier networks, while improving performance and functionality.

- **Principle 3:** *Provide data-centric query support:* In a sensor application the specific location (i.e. offset) of a record in a stream is unlikely to be of significance, except if it conveys information concerning the location and/or time at which the information was generated. We thus expect that applications will be best served by a query interface which allows them to locate data by value or attribute (e.g. location and time), rather than a read interface for unstructured data. This in turn implies the need to maintain metadata in the form of an index that provides low cost lookups.

2.4 System Design

TSAR embodies these design principles by employing local storage at sensors and a distributed index at the proxies. The key features of the system design are as follows:

In TSAR, writes occur at sensor nodes, and are assumed to consist of both opaque data as well as application-specific *metadata*. This metadata is a tuple of known types, which may be used by the application to locate and identify data records, and which may be searched on and compared by TSAR in the course of locating data for the application. In a camera-based sensing application, for instance, this metadata might include coordinates describing the field of view, average luminance, and motion values, in addition to basic information such as time and sensor location. Depending on the application, this metadata may be two or three orders of magnitude smaller than the data itself, for instance if the metadata consists of features extracted from image or acoustic data.

In addition to storing data locally, each sensor periodically sends a summary of reported metadata to a nearby proxy. The summary contains information such as the sensor ID, the interval (t_1, t_2) over which the summary was generated, a handle identifying the corresponding data record (e.g. its location in flash memory), and a coarse-grain representation of the metadata associated with the record. The precise data representation used in the summary is application-specific; for instance, a temperature sensor might choose to report the maximum and minimum temperature values observed in an interval as a coarse-grain representation of the actual time series.

The proxy uses the summary to construct an index; the index is global in that it stores information from all sensors in the system and it is distributed across the various proxies in the system. Thus, applications see a unified view of distributed data, and can query the index at any proxy to get access to data stored at any sensor. Specifically, each query triggers lookups in this distributed index and the list of matches is then used to retrieve the corresponding data from the sensors. There are several distributed index and lookup methods which might be used in this system; however, the index structure described in Section 3 is highly suited for the task.

Since the index is constructed using a coarse-grain summary, instead of the actual data, index lookups will yield approximate matches. The TSAR summarization mechanism guarantees that index lookups will never yield *false negatives* - i.e. it will never miss summaries which include the value being searched for. However, index lookups may yield *false positives*, where a summary matches the query but when queried the remote sensor finds no matching value, wasting network resources. The more coarse-grained the summary, the lower the update overhead and the greater the fraction of false positives, while finer summaries incur update overhead while reducing query overhead due to false positives. Remote sensors may easily distinguish false positives from queries which result in search hits, and calculate the ratio between the two; based on this ratio, TSAR employs a novel adaptive technique that dynamically varies the granularity of sensor summaries to balance the metadata overhead and the overhead of false positives.

3. Data Structures

At the proxy tier, TSAR employs a novel index structure called the Interval Skip Graph, which is an ordered, distributed data structure for finding all intervals that contain a particular point or range of values. Interval skip graphs combine Interval Trees [5], an interval-based binary search tree, with Skip Graphs [1], a ordered, distributed data structure for peer-to-peer systems [13]. The resulting data structure has two properties that make it ideal for sensor networks. First, it has $O(\log n)$ search complexity for accessing the first interval that matches a particular value or range, and constant complexity for accessing each successive interval. Second, indexing of intervals rather than individual values makes the data structure ideal for indexing summaries over time or value. Such summary-based indexing is a more natural fit for energy-constrained sensor nodes, since transmitting summaries incurs less energy overhead than transmitting all sensor data.

Definitions: We assume that there are N_p proxies and N_s sensors in a two-tier sensor network. Each proxy is responsible for multiple sensor nodes, and no assumption is made about the number of sensors per proxy. Each sensor transmits interval summaries of data or events regularly to one or more proxies that it is associated with, where interval i is represented as $[low_i, high_i]$. These intervals can correspond to time or value ranges that are used for indexing sensor data. No assumption is made about the size of an interval or about the amount of overlap between intervals.

Range queries on the intervals are posed by users to the network of proxies and sensors; each query q needs to determine all index values that overlap the interval $[low_q, high_q]$. The goal of the interval skip graph is to index all intervals such that the set that overlaps a query interval can be located efficiently. In the rest of this section, we describe the interval skip graph in greater detail.

3.1 Skip Graph Overview

In order to inform the description of the Interval Skip Graph, we first provide a brief overview of the Skip Graph data structure; for a more extensive description the reader is referred to [1]. Figure 2 shows a skip graph which indexes 8 keys; the keys may be seen along the bottom, and above each key are the pointers associated with that key. Each data element, consisting of a key and its associated pointers, may reside on a different node in the network,

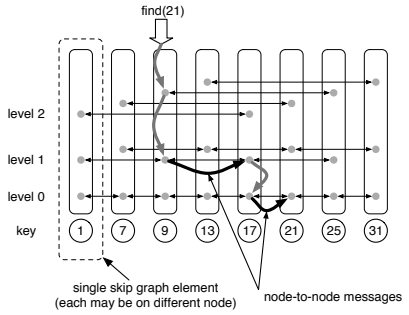


Figure 2: Skip Graph of 8 Elements

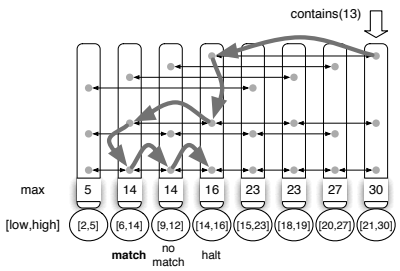


Figure 3: Interval Skip Graph

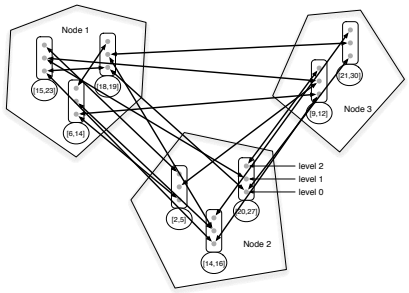


Figure 4: Distributed Interval Skip Graph

and pointers therefore identify both a remote node as well as a data element on that node. In this figure we may see the following properties of a skip graph:

- **Ordered index:** The keys are members of an ordered data type, for instance integers. Lookups make use of ordered comparisons between the search key and existing index entries. In addition, the pointers at the lowest level point directly to the successor of each item in the index.
- **In-place indexing:** Data elements remain on the nodes where they were inserted, and messages are sent between nodes to establish links between those elements and others in the index.
- **Log n height:** There are $\log_2 n$ pointers associated with each element, where n is the number of data elements indexed. Each pointer belongs to a level l in $[0...\log_2 n - 1]$, and together with some other pointers at that level forms a chain of $n/2^l$ elements.
- **Probabilistic balance:** Rather than relying on re-balancing operations which may be triggered at insert or delete, skip graphs implement a simple random balancing mechanism which maintains close to perfect balance on average, with an extremely low probability of significant imbalance.
- **Redundancy and resiliency:** Each data element forms an independent search tree root, so searches may begin at any node in the network, eliminating hot spots at a single search root. In addition the index is resilient against node failure; data on the failed node will not be accessible, but remaining data elements will be accessible through search trees rooted on other nodes.

In Figure 2 we see the process of searching for a particular value in a skip graph. The pointers reachable from a single data element form a binary tree: a pointer traversal at the highest level skips over $n/2$ elements, $n/4$ at the next level, and so on. Search consists of descending the tree from the highest level to level 0, at each level comparing the target key with the next element at that level and deciding whether or not to traverse. In the perfectly balanced case shown here there are $\log_2 n$ levels of pointers, and search will traverse 0 or 1 pointers at each level. We assume that each data element resides on a different node, and measure search cost by the number messages sent (i.e. the number of pointers traversed); this will clearly be $O(\log n)$.

Tree update proceeds from the bottom, as in a B-Tree, with the root(s) being promoted in level as the tree grows. In this way, for instance, the two chains at level 1 always contain $n/2$ entries each, and there is never a need to split chains as the structure grows. The update process then consists of choosing which of the 2^l chains to insert an element into at each level l, and inserting it in the proper place in each chain.

Maintaining a perfectly balanced skip graph as shown in Figure 2 would be quite complex; instead, the probabilistic balancing method introduced in Skip Lists [23] is used, which trades off a small amount of overhead in the expected case in return for simple update and deletion. The basis for this method is the observation that any element which belongs to a particular chain at level l can only belong to one of two chains at level $l+1$. To insert an element we ascend levels starting at 0, randomly choosing one of the two possible chains at each level, an stopping when we reach an empty chain.

One means of implementation (e.g. as described in [1]) is to assign each element an arbitrarily long random bit string. Each chain at level l is then constructed from those elements whose bit strings match in the first l bits, thus creating 2^l possible chains at each level and ensuring that each chain splits into exactly two chains at the next level. Although the resulting structure is not perfectly balanced, following the analysis in [23] we can show that the probability of it being significantly out of balance is extremely small; in addition, since the structure is determined by the random number stream, input data patterns cannot cause the tree to become imbalanced.

3.2 Interval Skip Graph

A skip graph is designed to store single-valued entries. In this section, we introduce a novel data structure that extends skip graphs to store intervals $[low_i, high_i]$ and allows efficient searches for all intervals covering a value v, i.e. $\{i : low_i \leq v \leq high_i\}$. Our data structure can be extended to range searches in a straightforward manner.

The interval skip graph is constructed by applying the method of augmented search trees, as described by Cormen, Leiserson, and Rivest [5] and applied to binary search trees to create an Interval Tree. The method is based on the observation that a search structure based on comparison of ordered keys, such as a binary tree, may also be used to search on a secondary key which is non-decreasing in the first key.

Given a set of intervals sorted by lower bound – $low_i \leq low_{i+1}$ – we define the secondary key as the cumulative maximum, $max_i = \max_{k=0...i}(high_k)$. The set of intervals intersecting a value v may then be found by searching for the first interval (and thus the interval with least low_i) such that $max_i \geq v$. We then

traverse intervals in increasing order lower bound, until we find the first interval with $low_i > v$, selecting those intervals which intersect v.

Using this approach we augment the skip graph data structure, as shown in Figure 3, so that each entry stores a range (lower bound and upper bound) and a secondary key (cumulative maximum of upper bound). To efficiently calculate the secondary key max_i for an entry i, we take the greatest of $high_i$ and the maximum values reported by each of i's left-hand neighbors.

To search for those intervals containing the value v, we first search for v on the secondary index, max_i, and locate the first entry with $max_i \geq v$. (by the definition of max_i, for this data element $max_i = high_i$.) If $low_i > v$, then this interval does not contain v, and no other intervals will, either, so we are done. Otherwise we traverse the index in increasing order of min_i, returning matching intervals, until we reach an entry with $min_i > v$ and we are done. Searches for all intervals which overlap a query range, or which completely contain a query range, are straightforward extensions of this mechanism.

Lookup Complexity: Lookup for the first interval that matches a given value is performed in a manner very similar to an interval tree. The complexity of search is $O(\log n)$. The number of intervals that match a range query can vary depending on the amount of overlap in the intervals being indexed, as well as the range specified in the query.

Insert Complexity: In an interval tree or interval skip list, the maximum value for an entry need only be calculated over the subtree rooted at that entry, as this value will be examined only when searching within the subtree rooted at that entry. For a simple interval skip graph, however, this maximum value for an entry must be computed over all entries preceding it in the index, as searches may begin anywhere in the data structure, rather than at a distinguished root element. It may be easily seen that in the worse case the insertion of a single interval (one that covers all existing intervals in the index) will trigger the update of all entries in the index, for a worst-case insertion cost of $O(n)$.

3.3 Sparse Interval Skip Graph

The final extensions we propose take advantage of the difference between the number of items indexed in a skip graph and the number of systems on which these items are distributed. The cost in network messages of an operation may be reduced by arranging the data structure so that most structure traversals occur locally on a single node, and thus incur zero network cost. In addition, since both congestion and failure occur on a per-node basis, we may eliminate links without adverse consequences if those links only contribute to load distribution and/or resiliency within a single node. These two modifications allow us to achieve reductions in asymptotic complexity of both update and search.

As may be in Section 3.2, insert and delete cost on an interval skip graph has a worst case complexity of $O(n)$, compared to $O(\log n)$ for an interval tree. The main reason for the difference is that skip graphs have a full search structure *rooted at each element*, in order to distribute load and provide resilience to system failures in a distributed setting. However, in order to provide load distribution and failure resilience it is only necessary to provide a full search structure *for each system*. If as in TSAR the number of nodes (proxies) is much smaller than the number of data elements (data summaries indexed), then this will result in significant savings.

Implementation: To construct a sparse interval skip graph, we ensure that there is a single distinguished element on each system, the *root element* for that system; all searches will start at one of these root elements. When adding a new element, rather than splitting lists at increasing levels l until the element is in a list with no others, we stop when we find that the element would be in a list containing no root elements, thus ensuring that the element is reachable from all root elements. An example of applying this optimization may be seen in Figure 5. (In practice, rather than designating existing data elements as roots, as shown, it may be preferable to insert null values at startup.)

When using the technique of membership vectors as in [1], this may be done by broadcasting the membership vectors of each root element to all other systems, and stopping insertion of an element at level l when it does not share an l-bit prefix with any of the N_p root elements. The expected number of roots sharing a $log_2 N_p$-bit prefix is 1, giving an expected expected height for each element of $log_2 N_p + O(1)$. An alternate implementation, which distributes information concerning root elements at pointer establishment time, is omitted due to space constraints; this method eliminates the need for additional messages.

Performance: In a (non-interval) sparse skip graph, since the expected height of an inserted element is now $\log_2 N_p + O(1)$, expected insertion complexity is $O(\log N_p)$, rather than $O(\log n)$, where N_p is the number of root elements and thus the number of separate systems in the network. (In the degenerate case of a single system we have a skip list; with splitting probability 0.5 the expected height of an individual element is 1.) Note that since searches are started at root elements of expected height $\log_2 n$, search complexity is not improved.

For an interval sparse skip graph, update performance is improved considerably compared to the $O(n)$ worst case for the non-sparse case. In an augmented search structure such as this, an element only stores information for nodes which may be reached from that element–e.g. the subtree rooted at that element, in the case of a tree. Thus, when updating the maximum value in an interval tree, the update is only propagated towards the root. In a sparse interval skip graph, updates to a node only propagate towards the N_p root elements, for a worst-case cost of $N_p \log_2 n$.

Shortcut search: When beginning a search for a value v, rather than beginning at the root on that proxy, we can find the element that is closest to v (e.g. using a secondary local index), and then begin the search at that element. The expected distance between this element and the search terminus is $\log_2 N_p$, and the search will now take on average $\log_2 N_p + O(1)$ steps. To illustrate this optimization, in Figure 4 depending on the choice of search root, a search for $[21, 30]$ beginning at node 2 may take 3 network hops, traversing to node 1, then back to node 2, and finally to node 3 where the destination is located, for a cost of 3 messages. The shortcut search, however, locates the intermediate data element on node 2, and then proceeds directly to node 3 for a cost of 1 message.

Performance: This technique may be applied to the primary key search which is the first of two insertion steps in an interval skip graph. By combining the short-cut optimization with sparse interval skip graphs, the expected cost of insertion is now $O(\log N_p)$, independent of the size of the index or the degree of overlap of the inserted intervals.

3.4 Alternative Data Structures

Thus far we have only compared the sparse interval skip graph with similar structures from which it is derived. A comparison with several other data structures which meet at least some of the requirements for the TSAR index is shown in Table 2.

Table 2: Comparison of Distributed Index Structures

	Range Query Support	Interval Representation	Re-balancing	Resilience	Small Networks	Large Networks
DHT, GHT	no	no	no	yes	good	good
Local index, flood query	yes	yes	no	yes	good	bad
P-tree, RP* (distributed B-Trees)	yes	possible	yes	no	good	good
DIMS	yes	no	yes	yes	yes	yes
Interval Skipgraph	yes	yes	no	yes	good	good

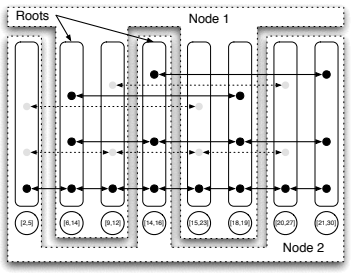


Figure 5: Sparse Interval Skip Graph

The hash-based systems, DHT [25] and GHT [26], lack the ability to perform range queries and are thus not well-suited to indexing spatio-temporal data. Indexing locally using an appropriate single-node structure and then flooding queries to all proxies is a competitive alternative for small networks; for large networks the linear dependence on the number of proxies becomes an issue. Two distributed B-Trees were examined - P-Trees [6] and RP* [19]. Each of these supports range queries, and in theory could be modified to support indexing of intervals; however, they both require complex re-balancing, and do not provide the resilience characteristics of the other structures. DIMS [18] provides the ability to perform spatio-temporal range queries, and has the necessary resilience to failures; however, it cannot be used index intervals, which are used by TSAR's data summarization algorithm.

4. Data Storage and Summarization

Having described the proxy-level index structure, we turn to the mechanisms at the sensor tier. TSAR implements two key mechanisms at the sensor tier. The first is a local archival store at each sensor node that is optimized for resource-constrained devices. The second is an adaptive summarization technique that enables each sensor to adapt to changing data and query characteristics. The rest of this section describes these mechanisms in detail.

4.1 Local Storage at Sensors

Interval skip graphs provide an efficient mechanism to lookup sensor nodes containing data relevant to a query. These queries are then routed to the sensors, which locate the relevant data records in the local archive and respond back to the proxy. To enable such lookups, each sensor node in TSAR maintains an archival store of sensor data. While the implementation of such an archival store is straightforward on resource-rich devices that can run a database, sensors are often power and resource-constrained. Consequently, the sensor archiving subsystem in TSAR is explicitly designed to exploit characteristics of sensor data in a resource-constrained setting.

Figure 6: Single storage record

Sensor data has very distinct characteristics that inform our design of the TSAR archival store. Sensors produce time-series data streams, and therefore, temporal ordering of data is a natural and simple way of storing archived sensor data. In addition to simplicity, a temporally ordered store is often suitable for many sensor data processing tasks since they involve time-series data processing. Examples include signal processing operations such as FFT, wavelet transforms, clustering, similarity matching, and target detection.

Consequently, the local archival store is a collection of records, designed as an append-only circular buffer, where new records are appended to the tail of the buffer. The format of each data record is shown in Figure 6. Each record has a metadata field which includes a timestamp, sensor settings, calibration parameters, etc. Raw sensor data is stored in the data field of the record. The data field is *opaque* and application-specific—the storage system does not know or care about interpreting this field. A camera-based sensor, for instance, may store binary images in this data field. In order to support a variety of applications, TSAR supports variable-length data fields; as a result, record sizes can vary from one record to another.

Our archival store supports three operations on records: create, read, and delete. Due to the append-only nature of the store, creation of records is simple and efficient. The create operation simply creates a new record and appends it to the tail of the store. Since records are always written at the tail, the store need not maintain a "free space" list. All fields of the record need to be specified at creation time; thus, the size of the record is known a priori and the store simply allocates the the corresponding number of bytes at the tail to store the record. Since writes are immutable, the size of a record does not change once it is created.

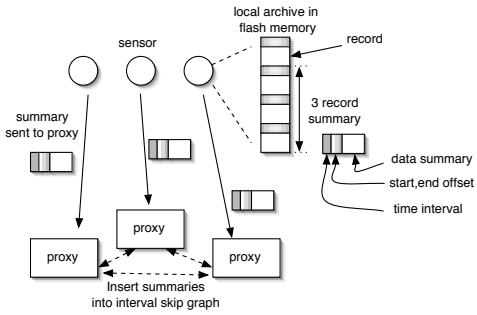


Figure 7: Sensor Summarization

The read operation enables stored records to be retrieved in order to answer queries. In a traditional database system, efficient lookups are enabled by maintaining a structure such as a B-tree that indexes certain keys of the records. However, this can be quite complex for a small sensor node with limited resources. Consequently, TSAR sensors do not maintain *any* index for the data stored in their archive. Instead, they rely on the proxies to maintain this metadata index—sensors periodically send the proxy information summarizing the data contained in a contiguous sequence of records, as well as a handle indicating the location of these records in flash memory.

The mechanism works as follows: In addition to the summary of sensor data, each node sends metadata to the proxy containing the time interval corresponding to the summary, as well as the start and end offsets of the flash memory location where the raw data corresponding is stored (as shown in Figure 7). Thus, random access is enabled at granularity of a summary—the start offset of each chunk of records represented by a summary is known to the proxy. Within this collection, records are accessed sequentially. When a query matches a summary in the index, the sensor uses these offsets to access the relevant records on its local flash by sequentially reading data from the start address until the end address. Any query-specific operation can then be performed on this data. Thus, no index needs to be maintained at the sensor, in line with our goal of simplifying sensor state management. The state of the archive is captured in the metadata associated with the summaries, and is stored and maintained at the proxy.

While we anticipate local storage capacity to be large, eventually there might be a need to overwrite older data, especially in high data rate applications. This may be done via techniques such as multi-resolution storage of data [9], or just simply by overwriting older data. When older data is overwritten, a delete operation is performed, where an index entry is deleted from the interval skip graph at the proxy and the corresponding storage space in flash memory at the sensor is freed.

4.2 Adaptive Summarization

The data summaries serve as glue between the storage at the remote sensor and the index at the proxy. Each update from a sensor to the proxy includes three pieces of information: the summary, a time period corresponding to the summary, and the start and end offsets for the flash archive. In general, the proxy can index the time interval representing a summary or the value range reported in the summary (or both). The former index enables quick lookups on all records seen during a certain interval, while the latter index enables quick lookups on all records matching a certain value.

As described in Section 2.4, there is a trade-off between the energy used in sending summaries (and thus the frequency and resolution of those summaries) and the cost of false hits during queries. The coarser and less frequent the summary information, the less energy required, while false query hits in turn waste energy on requests for non-existent data.

TSAR employs an adaptive summarization technique that balances the cost of sending updates against the cost of false positives. The key intuition is that each sensor can independently identify the fraction of false hits and true hits for queries that access its local archive. If most queries result in true hits, then the sensor determines that the summary can be coarsened further to reduce update costs without adversely impacting the hit ratio. If many queries result in false hits, then the sensor makes the granularity of each summary finer to reduce the number and overhead of false hits.

The resolution of the summary depends on two parameters—the interval over which summaries of the data are constructed and transmitted to the proxy, as well as the size of the application-specific summary. Our focus in this paper is on the interval over which the summary is constructed. Changing the size of the data summary can be performed in an application-specific manner (e.g. using wavelet compression techniques as in [9]) and is beyond the scope of this paper. Currently, TSAR employs a simple summarization scheme that computes the ratio of false and true hits and decreases (increases) the interval between summaries whenever this ratio increases (decreases) beyond a threshold.

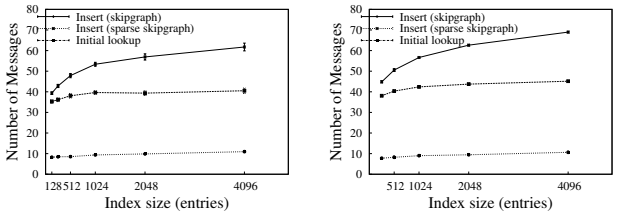
5. TSAR Implementation

We have implemented a prototype of TSAR on a multi-tier sensor network testbed. Our prototype employs Crossbow Stargate nodes to implement the proxy tier. Each Stargate node employs a 400MHz Intel XScale processor with 64MB RAM and runs the Linux 2.4.19 kernel and EmStar release 2.1. The proxy nodes are equipped with two wireless radios, a Cisco Aironet 340-based 802.11b radio and a hostmote bridge to the Mica2 sensor nodes using the EmStar transceiver. The 802.11b wireless network is used for inter-proxy communication within the proxy tier, while the wireless bridge enables sensor-proxy communication. The sensor tier consists of Crossbow Mica2s and Mica2dots, each consisting of a 915MHz CC1000 radio, a BMAC protocol stack, a 4 Mb on-board flash memory and an ATMega 128L processor. The sensor nodes run TinyOS 1.1.8. In addition to the on-board flash, the sensor nodes can be equipped with external MMC/SD flash cards using a custom connector. The proxy nodes can be equipped with external storage such as high-capacity compact flash (up to 4GB), 6GB micro-drives, or up to 60GB 1.8inch mobile disk drives.

Since sensor nodes may be several hops away from the nearest proxy, the sensor tier employs multi-hop routing to communicate with the proxy tier. In addition, to reduce the power consumption of the radio while still making the sensor node available for queries, low power listening is enabled, in which the radio receiver is periodically powered up for a short interval to sense the channel for transmissions, and the packet preamble is extended to account for the latency until the next interval when the receiving radio wakes up. Our prototype employs the MultiHopLEPSM routing protocol with the BMAC layer configured in the low-power mode with a 11% duty cycle (one of the default BMAC [22] parameters)

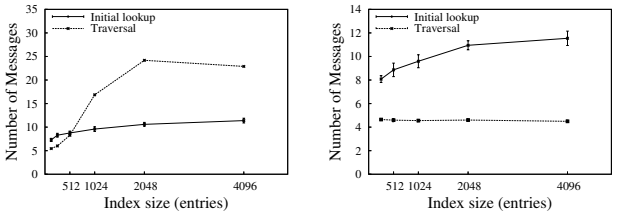
Our TSAR implementation on the Mote involves a data gathering task that periodically obtains sensor readings and logs these reading to flash memory. The flash memory is assumed to be a circular append-only store and the format of the logged data is depicted in Figure 6. The Mote sends a report to the proxy every N readings, summarizing the observed data. The report contains: (i) the address of the Mote, (ii) a handle that contains an offset and the length of the region in flash memory containing data referred to by the summary, (iii) an interval (t_1, t_2) over which this report is generated, (iv) a tuple $(low, high)$ representing the minimum and the maximum values observed at the sensor in the interval, and (v) a sequence number. The sensor updates are used to construct a sparse interval skip graph that is distributed across proxies, via network messages between proxies over the 802.11b wireless network.

Our current implementation supports queries that request records matching a time interval (t_1, t_2) or a value range (v_1, v_2). Spatial constraints are specified using sensor IDs. Given a list of matching intervals from the skip graph, TSAR supports two types of messages to query the sensor: lookup and fetch. A lookup message triggers a search within the corresponding region in flash memory and returns the number of matching records in that memory region (but does not retrieve data). In contrast, a fetch message not only



(a) James Reserve Data (b) Synthetic Data

Figure 8: Skip Graph Insert Performance



(a) James Reserve Data (b) Synthetic Data

Figure 9: Skip Graph Lookup Performance

triggers a search but also returns all matching data records to the proxy. Lookup messages are useful for polling a sensor, for instance, to determine if a query matches too many records.

6. Experimental Evaluation

In this section, we evaluate the efficacy of TSAR using our prototype and simulations. The testbed for our experiments consists of four Stargate proxies and twelve Mica2 and Mica2dot sensors; three sensors each are assigned to each proxy. Given the limited size of our testbed, we employ simulations to evaluate the behavior of TSAR in larger settings. Our simulation employs the EmTOS emulator [10], which enables us to run the same code in simulation and the hardware platform.

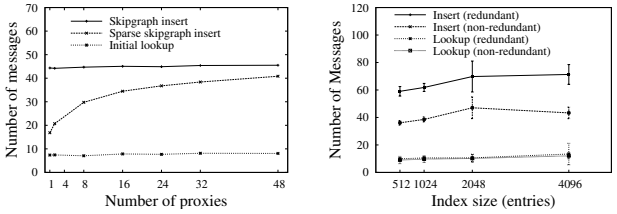
Rather than using live data from a real sensor, to ensure repeatable experiments, we seed each sensor node with a dataset (i.e., a trace) that dictates the values reported by that node to the proxy. One section of the flash memory on each sensor node is programmed with data points from the trace; these "observations" are then replayed during an experiment, logged to the local archive (located in flash memory, as well), and reported to the proxy. The first dataset used to evaluate TSAR is a temperature dataset from James Reserve [27] that includes data from eleven temperature sensor nodes over a period of 34 days. The second dataset is synthetically generated; the trace for each sensor is generated using a uniformly distributed random walk though the value space.

Our experimental evaluation has four parts. First, we run EmTOS simulations to evaluate the lookup, update and delete overhead for sparse interval skip graphs using the real and synthetic datasets. Second, we provide summary results from micro-benchmarks of the storage component of TSAR, which include empirical characterization of the energy costs and latency of reads and writes for the flash memory chip as well as the whole mote platform, and comparisons to published numbers for other storage and communication technologies. These micro-benchmarks form the basis for our full-scale evaluation of TSAR on a testbed of four Stargate proxies and twelve Motes. We measure the end-to-end query latency in our multi-hop testbed as well as the query processing overhead at the mote tier. Finally, we demonstrate the adaptive summarization capability at each sensor node. The remainder of this section presents our experimental results.

6.1 Sparse Interval Skip Graph Performance

This section evaluates the performance of sparse interval skip graphs by quantifying insert, lookup and delete overheads.

We assume a proxy tier with 32 proxies and construct sparse interval skip graphs of various sizes using our datasets. For each skip



(a) Impact of Number of Proxies (b) Impact of Redundant Summaries

Figure 10: Skip Graph Overheads

graph, we evaluate the cost of inserting a new value into the index. Each entry was deleted after its insertion, enabling us to quantify the delete overhead as well. Figure 8(a) and (b) quantify the insert overhead for our two datasets: each insert entails an initial traversal that incurs $\log n$ messages, followed by neighbor pointer update at increasing levels, incurring a cost of $4 \log n$ messages. Our results demonstrate this behavior, and show as well that performance of delete—which also involves an initial traversal followed by pointer updates at each level—incurs a similar cost.

Next, we evaluate the lookup performance of the index structure. Again, we construct skip graphs of various sizes using our datasets and evaluate the cost of a lookup on the index structure. Figures 9(a) and (b) depict our results. There are two components for each lookup—the lookup of the first interval that matches the query and, in the case of overlapping intervals, the subsequent linear traversal to identify all matching intervals. The initial lookup can be seen to takes $\log n$ messages, as expected. The costs of the subsequent linear traversal, however, are highly data dependent. For instance, temperature values for the James Reserve data exhibit significant spatial correlations, resulting in significant overlap between different intervals and variable, high traversal cost (see Figure 9(a)). The synthetic data, however, has less overlap and incurs lower traversal overhead as shown in Figure 9(b).

Since the previous experiments assumed 32 proxies, we evaluate the impact of the number of proxies on skip graph performance. We vary the number of proxies from 10 to 48 and distribute a skip graph with 4096 entries among these proxies. We construct regular interval skip graphs as well as sparse interval skip graphs using these entries and measure the overhead of inserts and lookups. Thus, the experiment also seeks to demonstrate the benefits of sparse skip graphs over regular skip graphs. Figure 10(a) depicts our results. In regular skip graphs, the complexity of insert is $O(log_2 n)$ in the

expected case (and $O(n)$ in the worst case) where n is the number of *elements*. This complexity is unaffected by changing the number of proxies, as indicated by the flat line in the figure. Sparse skip graphs require fewer pointer updates; however, their overhead is dependent on the number of proxies, and is $O(log_2 N_p)$ in the expected case, independent of n. This can be seen to result in significant reduction in overhead when the number of proxies is small, which decreases as the number of proxies increases.

Failure handling is an important issue in a multi-tier sensor architecture since it relies on many components—proxies, sensor nodes and routing nodes can fail, and wireless links can fade. Handling of many of these failure modes is outside the scope of this paper; however, we consider the case of resilience of skip graphs to proxy failures. In this case, skip graph search (and subsequent repair operations) can follow any one of the other links from a root element. Since a sparse skip graph has search trees rooted at each node, searching can then resume once the lookup request has routed around the failure. Together, these two properties ensure that even if a proxy fails, the remaining entries in the skip graph will be reachable with high probability—only the entries on the failed proxy and the corresponding data at the sensors becomes inaccessible.

To ensure that all data on sensors remains accessible, even in the event of failure of a proxy holding index entries for that data, we incorporate redundant index entries. TSAR employs a simple redundancy scheme where additional coarse-grain summaries are used to protect regular summaries. Each sensor sends summary data periodically to its local proxy, but less frequently sends a lower-resolution summary to a backup proxy—the backup summary represents all of the data represented by the finer-grained summaries, but in a lossier fashion, thus resulting in higher read overhead (due to false hits) if the backup summary is used. The cost of implementing this in our system is low – Figure 10(b) shows the overhead of such a redundancy scheme, where a single coarse summary is send to a backup for every two summaries sent to the primary proxy. Since a redundant summary is sent for every two summaries, the insert cost is 1.5 times the cost in the normal case. However, these redundant entries result in only a negligible increase in lookup overhead, due the logarithmic dependence of lookup cost on the index size, while providing full resilience to any single proxy failure.

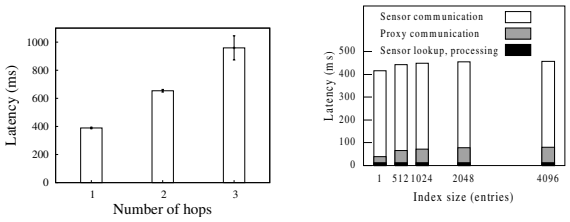
6.2 Storage Microbenchmarks

Since sensors are resource-constrained, the energy consumption and the latency at this tier are important measures for evaluating the performance of a storage architecture. Before performing an end-to-end evaluation of our system, we provide more detailed information on the energy consumption of the storage component used to implement the TSAR local archive, based on empirical measurements. In addition we compare these figures to those for other local storage technologies, as well as to the energy consumption of wireless communication, using information from the literature. For empirical measurements we measure energy usage for the storage component itself (i.e. current drawn by the flash chip), as well as for the entire Mica2 mote.

The power measurements in Table 3 were performed for the AT45DB041 [15] flash memory on a Mica2 mote, which is an older NOR flash device. The most promising technology for low-energy storage on sensing devices is NAND flash, such as the Samsung K9K4G08U0M device [16]; published power numbers for this device are provided in the table. Published energy requirements for wireless transmission using the Chipcon [4] CC2420 radio (used in MicaZ and Telos motes) are provided for comparison, assuming

	Energy	Energy/byte
Mote flash		
Read 256 byte page	58μJ* / 136μJ* total	0.23μJ*
Write 256 byte page	926μJ* / 1042μJ* total	3.6μJ*
NAND Flash		
Read 512 byte page	2.7μJ	1.8nJ
Write 512 byte page	7.8μJ	15nJ
Erase 16K byte sector	60μJ	3.7nJ
CC2420 radio		
Transmit 8 bits (-25dBm)	0.8μJ	0.8μJ
Receive 8 bits	1.9μJ	1.9μJ
Mote AVR processor		
In-memory search, 256 bytes	1.8μJ	6.9nJ

Table 3: Storage and Communication Energy Costs (*measured values)



(a) Multi-hop query performance

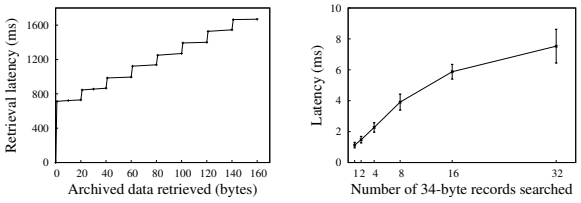
(b) Query Performance

Figure 11: Query Processing Latency

zero network and protocol overhead. Comparing the total energy cost for writing flash (erase + write) to the total cost for communication (transmit + receive), we find that the NAND flash is almost 150 times more efficient than radio communication, even assuming perfect network protocols.

6.3 Prototype Evaluation

This section reports results from an end-to-end evaluation of the TSAR prototype involving both tiers. In our setup, there are four proxies connected via 802.11 links and three sensors per proxy. The multi-hop topology was preconfigured such that sensor nodes were connected in a line to each proxy, forming a minimal tree of depth



(a) Data Query and Fetch Time

(b) Sensor query processing delay

Figure 12: Query Latency Components

3. Due to resource constraints we were unable to perform experiments with dozens of sensor nodes, however this topology ensured that the network diameter was as large as for a typical network of significantly larger size.

Our evaluation metric is the end-to-end latency of query processing. A query posed on TSAR first incurs the latency of a sparse skip graph lookup, followed by routing to the appropriate sensor node(s). The sensor node reads the required page(s) from its local archive, processes the query on the page that is read, and transmits the response to the proxy, which then forwards it to the user. We first measure query latency for different sensors in our multi-hop topology. Depending on which of the sensors is queried, the total latency increases almost linearly from about 400ms to 1 second, as the number of hops increases from 1 to 3 (see Figure 11(a)).

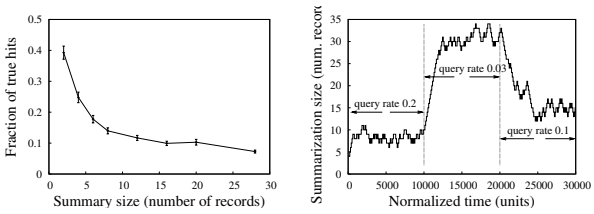
Figure 11(b) provides a breakdown of the various components of the end-to-end latency. The dominant component of the total latency is the communication over one or more hops. The typical time to communicate over one hop is approximately 300ms. This large latency is primarily due to the use of a duty-cycled MAC layer; the latency will be larger if the duty cycle is reduced (e.g. the 2% setting as opposed to the 11.5% setting used in this experiment), and will conversely decrease if the duty cycle is increased. The figure also shows the latency for varying index sizes; as expected, the latency of inter-proxy communication and skip graph lookups increases logarithmically with index size. Not surprisingly, the overhead seen at the sensor is independent of the index size.

The latency also depends on the number of packets transmitted in response to a query—the larger the amount of data retrieved by a query, the greater the latency. This result is shown in Figure 12(a). The step function is due to packetization in TinyOS; TinyOS sends one packet so long as the payload is smaller than 30 bytes and splits the response into multiple packets for larger payloads. As the data retrieved by a query is increased, the latency increases in steps, where each step denotes the overhead of an additional packet.

Finally, Figure 12(b) shows the impact of searching and processing flash memory regions of increasing sizes on a sensor. Each summary represents a collection of records in flash memory, and all of these records need to be retrieved and processed if that summary matches a query. The coarser the summary, the larger the memory region that needs to be accessed. For the search sizes examined, amortization of overhead when searching multiple flash pages and archival records, as well as within the flash chip and its associated driver, results in the appearance of sub-linear increase in latency with search size. In addition, the operation can be seen to have very low latency, in part due to the simplicity of our query processing, requiring only a compare operation with each stored element. More complex operations, however, will of course incur greater latency.

6.4 Adaptive Summarization

When data is summarized by the sensor before being reported to the proxy, information is lost. With the interval summarization method we are using, this information loss will never cause the proxy to believe that a sensor node does not hold a value which it in fact does, as all archived values will be contained within the interval reported. However, it does cause the proxy to believe that the sensor may hold values which it does not, and forward query messages to the sensor for these values. These *false positives* constitute the cost of the summarization mechanism, and need to be balanced against the savings achieved by reducing the number of reports. The goal of adaptive summarization is to dynamically vary the summary size so that these two costs are balanced.



(a) Impact of summary size

(b) Adaptation to query rate

Figure 13: Impact of Summarization Granularity

Figure 13(a) demonstrates the impact of summary granularity on false hits. As the number of records included in a summary is increased, the fraction of queries forwarded to the sensor which match data held on that sensor ("true positives") decreases. Next, in Figure 13(b) we run the a EmTOS simulation with our adaptive summarization algorithm enabled. The adaptive algorithm increases the summary granularity (defined as the number of records per summary) when $\frac{Cost(updates)}{Cost(falsehits)} > 1 + \epsilon$ and reduces it if $\frac{Cost(updates)}{Cost(falsehits)} > 1 - \epsilon$, where ϵ is a small constant. To demonstrate the adaptive nature of our technique, we plot a time series of the summarization granularity. We begin with a query rate of 1 query per 5 samples, decrease it to 1 every 30 samples, and then increase it again to 1 query every 10 samples. As shown in Figure 13(b), the adaptive technique adjusts accordingly by sending more fine-grain summaries at higher query rates (in response to the higher false hit rate), and fewer, coarse-grain summaries at lower query rates.

7. Related Work

In this section, we review prior work on storage and indexing techniques for sensor networks. While our work addresses both problems jointly, much prior work has considered them in isolation.

The problem of archival storage of sensor data has received limited attention in sensor network literature. ELF [7] is a log-structured file system for local storage on flash memory that provides load leveling and Matchbox is a simple file system that is packaged with the TinyOS distribution [14]. Both these systems focus on local storage, whereas our focus is both on storage at the remote sensors as well as providing a unified view of distributed data across all such local archives. Multi-resolution storage [9] is intended for in-network storage and search in systems where there is significant data in comparison to storage resources. In contrast, TSAR addresses the problem of archival storage in two-tier systems where sufficient resources can be placed at the edge sensors. The RISE platform [21] being developed as part of the NODE project at UCR addresses the issues of hardware platform support for large amounts of storage in remote sensor nodes, but not the indexing and querying of this data.

In order to efficiently access a distributed sensor store, an index needs to be constructed of the data. Early work on sensor networks such as Directed Diffusion [17] assumes a system where all useful sensor data was stored locally at each sensor, and spatially scoped queries are routed using geographic co-ordinates to locations where the data is stored. Sources publish the events that they detect, and sinks with interest in specific events can subscribe to these events. The Directed Diffusion substrate routes queries to specific locations

if the query has geographic information embedded in it (e.g.: find temperature in the south-west quadrant), and if not, the query is flooded throughout the network.

These schemes had the drawback that for queries that are not geographically scoped, search cost ($O(n)$ for a network of n nodes) may be prohibitive in large networks with frequent queries. Local storage with in-network indexing approaches address this issue by constructing indexes using frameworks such as Geographic Hash Tables [24] and Quad Trees [9]. Recent research has seen a growing body of work on data indexing schemes for sensor networks[26][11][18]. One such scheme is DCS [26], which provides a hash function for mapping from event name to location. DCS constructs a distributed structure that groups events together spatially by their named type. Distributed Index of Features in Sensornets (DIFS [11]) and Multi-dimensional Range Queries in Sensor Networks (DIM [18]) extend the data-centric storage approach to provide spatially distributed hierarchies of indexes to data.

While these approaches advocate in-network indexing for sensor networks, we believe that indexing is a task that is far too complicated to be performed at the remote sensor nodes since it involves maintaining significant state and large tables. TSAR provides a better match between resource requirements of storage and indexing and the availability of resources at different tiers. Thus complex operations such as indexing and managing metadata are performed at the proxies, while storage at the sensor remains simple.

In addition to storage and indexing techniques specific to sensor networks, many distributed, peer-to-peer and spatio-temporal index structures are relevant to our work. DHTs [25] can be used for indexing events based on their type, quad-tree variants such as R-trees [12] can be used for optimizing spatial searches, and K-D trees [2] can be used for multi-attribute search. While this paper focuses on building an ordered index structure for range queries, we will explore the use of other index structures for alternate queries over sensor data.

8. Conclusions

In this paper, we argued that existing sensor storage systems are designed primarily for flat hierarchies of homogeneous sensor nodes and do not fully exploit the multi-tier nature of emerging sensor networks. We presented the design of *TSAR*, a fundamentally different storage architecture that envisions separation of data from metadata by employing local storage at the sensors and distributed indexing at the proxies. At the proxy tier, TSAR employs a novel multi-resolution ordered distributed index structure, the Sparse Interval Skip Graph, for efficiently supporting spatio-temporal and range queries. At the sensor tier, TSAR supports energy-aware adaptive summarization that can trade-off the energy cost of transmitting metadata to the proxies against the overhead of false hits resulting from querying a coarser resolution index structure. We implemented TSAR in a two-tier sensor testbed comprising Stargate-based proxies and Mote-based sensors. Our experimental evaluation of TSAR demonstrated the benefits and feasibility of employing our energy-efficient low-latency distributed storage architecture in multi-tier sensor networks.

9. REFERENCES

[1] James Aspnes and Gauri Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Baltimore, MD, USA, 12–14 January 2003.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management.*, January 2001.

[4] Chipcon. CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF transceiver, 2004.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition edition, 2001.

[6] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. Technical Report TR2004-1926, Cornell University, 2004.

[7] Hui Dai, Michael Neufeld, and Richard Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 176–187, New York, NY, USA, 2004. ACM Press.

[8] Peter Desnoyers, Deepak Ganesan, Huan Li, and Prashant Shenoy. PRESTO: A predictive storage architecture for sensor networks. In *Tenth Workshop on Hot Topics in Operating Systems (HotOS X).*, June 2005.

[9] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution storage in sensor networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys).*, 2003.

[10] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.

[11] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A distributed index for features in sensor networks. *Elsevier Journal of Ad Hoc Networks*, 2003.

[12] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[13] Nicholas Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *In proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[14] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[15] Atmel Inc. 4-megabit 2.5-volt or 2.7-volt DataFlash AT45DB041B, 2005.

[16] Samsung Semiconductor Inc. K9W8G08U1M, K9K4G08U0M: 512M x 8 bit / 1G x 8 bit NAND flash memory, 2003.

[17] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, August 2000. ACM Press.

[18] Xin Li, Young-Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys).*, 2003. to appear.

[19] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. RP*: A family of order preserving scalable distributed data structures. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 342–353, San Francisco, CA, USA, 1994.

[20] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, Boston, MA, 2002.

[21] A. Mitra, A. Banerjee, W. Najjar, D. Zeinalipour-Yazti, D.Gunopulos, and V. Kalogeraki. High performance, low power sensor platforms featuring gigabyte scale storage. In *SenMetrics 2005: Third International Workshop on Measurement, Modeling, and Performance Analysis of Wireless Sensor Networks*, July 2005.

[22] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[23] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[24] S. Ratnasamy, D. Estrin, R. Govindan, B. Karp, L. Yin S. Shenker, and F. Yu. Data-centric storage in sensornets. In *ACM First Workshop on Hot Topics in Networks*, 2001.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[26] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT - a geographic hash-table for data-centric storage. In *First ACM International Workshop on Wireless Sensor Networks and their Applications*, 2002.

[27] N. Xu, E. Osterweil, M. Hamilton, and D. Estrin. `http://www.lecs.cs.ucla.edu/~nxu/ess/`. James Reserve Data.