Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2006-077 November 27, 2006

Scoop: An Adaptive Indexing Scheme for Stored Data in Sensor Networks

Thomer M. Gil and Samuel Madden

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Scoop: An Adaptive Indexing Scheme for Stored Data in Sensor Networks

Thomer M. Gil and Samuel Madden
{thomer,madden}@csail.mit.edu

Abstract

In this paper, we present the design of Scoop, a system for indexing and querying stored data in sensor networks. Scoop works by collecting statistics about the rate of queries and distribution of sensor readings over a sensor network, and uses those statistics to build an index that tells nodes where in the network to store their readings. Using this index, a user's queries over that stored data can be answered efficiently, without flooding those queries throughout the network. This approach offers a substantial advantage over other solutions that either store all data externally on a basestation (requiring every reading to be collected from all nodes), or that store all data locally on the node that produced it (requiring queries to be flooded throughout the network). Our results, in fact, show that Scoop offers a factor of four improvement over existing techniques in a real implementation on a 64-node mote-based sensor network. These results also show that Scoop is able to efficiently adapt to changes in the distribution and rates of data and queries.

1 Introduction

Sensor networks offer the promise of fine-granularity, low-cost data collection from difficult-to-reach remote environments. Existing data collection tools (like Cougar [24] and TinyDB [17]) as well as many deployments(e.g, [2, 6, 18]) have demonstrated the potential of this new technology. However, these existing approaches all tend to work by picking a sample rate and delivering all data from the network to some "root" node where the user receives data at that pre-selected rate.

In contrast, we have developed a system called Scoop that allows users to "scoop" up sensor readings of particular interest to them. As in other systems, nodes continuously sample data, but rather than immediately transmitting data to the root, Scoop nodes use an adaptive *storage index* (which is centrally generated by the root, as we explain below) that tells them where to store data – either locally in their Flash memories, in the Flash memory of a nearby node, or perhaps even on the basestation. Users then query for readings that satisfy conditions of particular interest – in certain time or value ranges, for example. Queries can be answered efficiently by using the storage index to determine which nodes have a particular value. This allows the network to transmit far fewer packets than in existing systems when queries are relatively infrequent (since most data doesn't have to be transmitted to the root). When queries are frequent, Scoop performs as well as these existing systems because the storage index is adapted to cause data to be sent directly to the root.

There are a number of applications where a Scoop-like in-network indexing system is useful. Consider, for example, a sensornet deployed for monitoring a factory floor that uses sensors on equipment to measure temperature or vibrational energy in a certain frequency band. Real-world examples of such deployments (e.g., [2]) typically consist of some number of battery powered nodes on different pieces of equipment. (Batteries obviate the need for expensive and possibly dangerous power wires.) Current deployments (like [2]) typically send all sensor readings to a centralized basestation for analysis, but a more power-efficient approach would be to collect readings on the nodes, possibly pre-process them locally, and store the values at or near the detecting nodes in the network. Users could then query the history of readings relevant to their interests.

Making a Scoop-like index work efficiently is tricky, because the multihop nature of sensor networks means that locating data that satisfies arbitrary query predicates inside the network is hard. In a naive system, it might require flooding the network, interrogating each node to see if it has data satisfying a particular request. In Scoop, we address this challenge by using a novel statistics-based approach, which works as follows: periodically, nodes report to the basestation histograms summarizing the data they have produced recently. The basestation aggregates these histograms together to produce a storage index that maps sensor values to nodes in the network that should store those values (here, values could be, for example, simple temperature readings or possibly more complex values that are the outcome of some computation over one or more attributes.) This mapping tells nodes where to store individual data readings. The basestation disseminates this storage index throughout the network and uses it to answer queries.

As nodes produce data, they use the mapping to determine where that data should be stored. When the basestation builds the mapping, it uses an indexing algorithm (based on a simple optimization problem) that attempts to place data items near to sensors that are likely to produce that data (based on historical patterns of data production), as well as placing popular data items (based on historical queries) to be stored closer to the basestation. Because nodes typically produce similar values over time, and because there is typically some geographic locality between values produced by nodes, nodes often are assigned their own values in the mapping, and when they aren't, values often only have to to be transmitted one or two network hops. This leads to dramatically better results than transmitting all data to a single basestation, which quickly becomes a network bottleneck in systems like TinyDB [17]. Our results show that Scoop typically is able to use about a quarter of the transmissions (under a reasonable query workload) when compared to simple schemes collect all data to a basestation (such as Cougar and TinyDB), despite the additional overheads of statistics collection and storage index dissemination.

Unlike traditional database indices, Scoop storage indices *adapt* over time, placing particular sensor values at different locations in the network over time. Because statistics are reported periodically, the sink is able to recompute the storage index and re-disseminate it when it has changed significantly. When a query for data from a particular point in time arrives, the basestation can compute which storage index was in effect at that time and request that the nodes with the appropriate data be used to answer the query.

We have built a complete implementation of Scoop for TinyOS-based motes [4] and evaluated its performance on a 62-node testbed and in the TinyOS simulator, TOSSIM. We show that our system scales well up to a few hundred nodes, which is comparable to the size of the largest single-basestation sensor network deployments in use today [6, 2]. This paper describes the Scoop system, focusing on the algorithms and protocols for statistics collection, storage index creation and dissemination, and query answer, as well as describing our experimental results and highlighting some directions for future research.

2 Background

2.1 Hardware Trends

Storage: Current-generation hardware has a small amount of RAM (a few KB to 10's of KB) with a significantly larger amount (up to several MB) of (non-volatile) Flash memory, where Scoop stores its intermediate query results. Future generations of devices will certainly have both more RAM and Flash, particularly as consumer devices like digital cameras and MP3 players continue to drive the commoditization of very low-power, high capacity Flash memories.

Communication: In wireless sensor networks (WSNs), radio communication tends to be quite lossy without retransmission; motes drop significant numbers of packets. Though retransmission can mitigate these losses somewhat, nodes can still fail, move away, or be subject to radio interference that makes them temporarily unable to communicate with some or all of their neighbors. The radios on the Mica2 motes we use provide a single, shared, 38.6 kilobits per second (Kbps) communication channel. The actual, usable application bandwidth is closer to 10 Kbps once channel access and packet-header overheads are figured in. Newer 802.15.4 have a maximum raw bandwidth of 250 Kbps; delivered throughput will be closer to 100 Kbps.

Power: Because sensors are battery powered, power consumption is of utmost concern to application designers. Power is consumed by a number of factors; typically communications dominates this cost [16, 20]. Our previous work [16] suggests that up to 90% of the energy consumption of a typical data collection system for sensor networks is due to communication. Current trends suggest that the cost-per-bit of radio transmission will continue to dominate the cost to store and retrieve data from memory—even relatively power-hungry non-volatile Flash. For example, it costs about 28 nJ to write 1 bit to a current-generation Micron Technology 128 Mbit NX25P32 Flash chip. Reads are substantially cheaper. In contrast, current generation 802.15.4 radios consume about 15 mJ of power per second, for a total energy consumption of about 700 nJ/bit, making radio about two orders of magnitude more expensive than Flash per transmitted (or stored) bit.

2.2 Software

Motes run a basic operating system called TinyOS [12], which provides a suite of software libraries for sending and receiving messages, organizing motes into ad-hoc, multihop routing trees, storing data to and from Flash and acquiring data from sensors. In this section, we briefly summarize the features of TinyOS that are salient to the design of Scoop.

TinyOS provides a simple link-layer that allows nodes to exchange messages with other nodes that are within radio range. Multiple nodes that want to send messages concurrently negotiate channel access using CSMA-CA, a variant of the protocol used in shared Ethernet.

The most common multihop networking protocol in WSNs is *tree-based routing*. Tree-based routing organizes the nodes in the network into a spanning tree rooted at some *basestation* node at the root of the tree. This tree allows the basestation to collect data from or disseminate data to all of the nodes in a network. The basic idea is to repeatedly broadcast a *tree-join* message from the root down the tree. Nodes pick as their parent one of the nodes from which they heard the *tree-join* message. The practical details of such protocols are covered work by Woo et al. [23] and DeCouto et al. [3]; we use an implementation based on code by Woo et al. [23].

3 Data and Query Model

Scoop operates on a network of nodes that sample data and store data at a certain *sample rate*. Periodically, the user issues queries over this data from a basestation. Queries consist of a range of values or a list of nodes to be queried, which are equivalent to queries of the form:

```
SELECT attr_1 ... attr_n
  FROM sensors
  WHERE pred_1 ... pred_n
```

In this work, we focus on this kind of snapshot queries that retrieve the value of one or more attributes that are indexed by a Scoop index.

Scoop exports a simple, attribute-based data model based on TinyDB [17]. Each attribute provides a function to sample the sensor(s) representing the attribute, and Scoop invokes this function at the user-specified rate to generate data to be stored. This sampling rate is currently established at compile time. This attribute interface currently supports temperature, humidity, light, acceleration, and sound volume sensors.

In the next two sections, we describe how Scoop implements this query interface to stored data. We focus on creating the storage index (Section 4), efficiently collecting relevant statistics needed for doing that creation, (Section 5.2) efficiently disseminating the storage index (Section 5.3), routing data using the index (Section 5.4), and, finally, on answering queries (Section 5.5).

4 Storage Indices

This section motivates the design of Scoop's storage index. We begin by describing two existing (non-index based) storage techniques for store-and-query sensor networks into two categories: (note that a similar taxonomy of storage policies appears in [21], but they do not evaluate an adaptive, statistics driven approach like Scoop.)

One possible storage technique is "send-to-base": sensors send all their data to the basestation through a network routing tree rooted at the basestation. As mentioned above, this can be wasteful, since energy is spent sending data to the basestation where it might never be used. Secondly, depending on data rates, the network may become saturated if all sensors try to send data simultaneously, resulting in high loss.

A second approach is "store-local": sensors store sampled data locally. The basestation floods queries through the entire network; sensors send their reply back. Unfortunately, this is expensive since only a fraction of the sensors may actually have relevant data. In contrast to send-to-base, store-local is efficient when data rates are much higher than the query rate.

Scoop, in contrast, adapts between the two extremes of send-to-base and store-local: data is stored closer to the basestation when the query rate is higher than data rates, and data is stored closer to the source when data rates are higher than query rates. Each value is stored on a specific node, as specified in a *storage index* that is periodically updated by the basestation and then broadcast to all nodes.

A storage index is a value to node ID mapping. In this paper we simply map attributes ranges to node ID, as illustrated in Figure 1, which shows an simple example temperature storage index for time period T1-T2. The node on the right hand side is responsible for storing all temperature readings in the left column, during T1-T2. Nodes may have multiple non-overlapping ranges assigned to them, like node 2.

Temperature time: T1-T2	
values	*node*
20-22	2
23-26	1
27-28	5
⋮	⋮
34-36	2

Figure 1. A storage index for temperature.

One such mapping exists per indexed attribute, per time period. The basestation creates a storage index based on statistics over the previous few minutes/seconds. (In our experiments, the basestation recreates a new storage index every 4 minutes). The mapping is chosen to minimize the total number of messages the system sends, as described below. This approach relies on the insight that recently sensed values are likely to be a good predictor of values a node produces in the near future; this temporal correlation has been shown to be present in practice in sensor data in several recent papers on the use of statistical models for sensor value prediction [9, 5]. Intuitively, it makes sense that there would be some stationarity in many kinds of sensor data; for example, in vibrating equipment, the amplitude and frequency of vibration is likely to remain roughly the same over a short time window on a particular piece of equipment.

Clearly, the particular index that is chosen impacts the communication overhead. For example, assigning a value that is queried very frequently to a location far away from the basestation will result in high query/reply overhead. Storing the value on a node closer to the basestation reduces this overhead, but now the cost of sending messages from nodes that produce this value to the chosen destination node may increase. Similarly, mapping a value v to a node p that is more likely to produce v reduces the overhead of sending p's data.

Our algorithm for selecting a storage index is guided by the following properties that a communication-efficient storage index should have:

- **P1:** In the absence of other changes, if the data rate goes up, data should be stored closer to the source (or the source itself) to avoid sending that data across many hops.
- **P2:** In the absence of other changes, if the query rate goes up, data should be stored closer to the basestation to avoid sending queries and replies across many hops.
- **P3:** In the absence of other changes data should be stored closest to the location where it is most likely going to be produced.
- **P4:** The storage index should take network conditions into account to avoid, for example, forcing a node to send data to another node over a lossy link, causing expensive retransmissions.

The algorithm the basestation runs periodically to find a storage index is outlined in Figure 2. The goal is to find one *owner*, o, for each value, v, i.e., the node that is responsible for storing all readings of v. The set of value-to-node mappings is the storage index. This algorithm tries to pick an owner that satisfies the minimum expected messages metric described above, by placing a given sensor value (or class of value) on the node that will incur the minimum overall number of transmissions over time. (Section 5.2 discusses how the basestation obtains the various statistics needed in this algorithm.)

```
for all values: v {              [v = value]
  for all sensors: o {           [o = owner]
    for all sensors: p {         [p = producer of v]
      cost(o,v) += P(p produces v) × rate_p ×
        xmits(p → o)
    }
    cost(o,v) += P(user queries v) × query rate ×
      xmits(base → o → base)
  }
  storage_index[v] = argmin_o (cost(o,v))
}
```

rate$_x$: the rate at which node x produces data
P(X): the probability that X happens
xmits$(x \rightarrow y)$: the estimated number of transmissions required to get a packet from x to y.
cost(o,v): expected no. of msgs. if v stored at node o

Figure 2. Indexing algorithm.

The outer loop iterates over all possible values v of the attribute to find an owner for it by simply trying out all possible nodes as owner (the second loop) and picking the best one. For each potential owner, o, it computes the cost (i.e., number of messages) if that node were the owner of v. (The current version of Scoop computes the cost in terms of number of messages, but the algorithm could easily include power consumption, storage capacity on nodes, the expected reply volume, or even the cost associated with disseminating the storage index itself in the cost metric.) The cost is twofold: sending v from all sensors that produce it to o (innermost loop) plus querying o from the basestation. The former is the product of the probability that each node p produces value v, the rate at which it does this, and the expected cost of sending data from p to o. Similarly, the cost to query node o is the product of the probability that a user issues a query about value v, the query rate, and the expected number of transmissions to send the query from the basestation to o and back. The best owner for a value v is the one that minimizes this cost.

This algorithm satisfies the aforementioned properties. **P1:** if the data rate of p goes up, $cost(o)$ goes up for all o's far away from p; hence, a node closer to p (or p itself) will be better. **P2:** if the query rate goes up, $cost(o)$ goes up for all o's further away from the basestation; hence a node closer to the basestation will be better. **P3:** the more likely it is that a certain node p produces v, the more attractive it is to pick p (or a node closer to p) as owner for v because of the lower transmission cost. **P4:** the expected number of transmissions, i.e., xmits($x \rightarrow y$), takes network connectivity into account; the basestation uses statistics it collects from the nodes as discussed in Section 5.2.

The time-complexity of this algorithm is $O(Vn^2)$, where n is the number of nodes and V is the number of values in the domain of the attribute. In our experiments that used real sensor traces, V was at about 150 and n was 62. For the size of sensor networks we are aiming for – a few hundred nodes – this algorithm is very practical.

Notice that this algorithm may generate a "send-to-base" policy (if all values get mapped to the basestation), but never a "store-local" policy (since the current version never maps overlapping ranges to more than one node). The basestation, therefore, also evaluates the expected cost of a "store-local" storage index and uses it if the expected cost is lower than the cost of the best storage index.

Extensions: Though we focus on integer values of a single attribute, values can also represent more complex composite detections. For example, in an industrial monitoring network, each sensor might classify its last few sensor readings according to their vibration level on a scale of 1-20, and the mapping might tell the sensor where to store a particular class of vibrations.

Another extension of this algorithm is to pick multiple owners, i.e., an owner *set*, per value, thus allowing nodes to pick one nearby node from multiple owner candidates to store their data. Having multiple owners per value may reduce communication overhead if multiple regions in the network exhibit similar data distributions. However, it may increase the size of a storage index and the cost for querying that value. Naively considering all possible owner sets makes the algorithm's time-complexity exponential in n. Hence, a more feasible approach is to consider only small owner sets.

A third extension involves range queries. Rather than considering the placement of individual values, we could consider placing ranges of values (e.g., modify the outer loop of the placement algorithm to consider a fixed set of ranges rather than a fixed set of values). The challenge is choosing which ranges to iterate over: we might use distinct ranges that have appeared in queries, or simply fixed segmentation of values (e.g., 10 degree temperature ranges.) The advantage of placing ranges is that popular range queries can likely be satisfied by just going to one or a small number of nodes; a potential disadvantage is that large ranges may end up being stored on a single node, which could increase the storage burden on that node.

5 Scoop Design

Given this basic algorithm for index creation, we now describe how scoop collects statistics, disseminates indices, routes sensor readings, and answers queries.

5.1 Routing tree

Nodes collectively build and maintain a routing tree of the sort commonly used in sensor networks. This allows Scoop to route packets to the basestation. The routing tree spans the network and is formed by having each node select exactly one *parent* that is one-hop closer to the basestation than itself.

A node maintains a "descendants list" of all its children, children's children, and so on, by tracking all nodes on whose behalf it routes packets up the routing tree. This list contains at most n entries (32, in our experiments) and is used for routing data (Section 5.4) and routing queries (Section 5.5). Finally, each node keeps track of the nodes in its direct network neighborhood, independent of the routing tree. This list, too, has a maximum size (32, in our experiments) and is used to optimize routing. A node evicts other nodes from its lists after not hearing from them for a long time, thus adapting to changes in network connectivity. If a node has more than n descendants, the routing algorithm will still work, though with somewhat degraded performance (see Section 5.4.)

5.2 Statistics Collection

The basestation relies on various statistics to run the storage index algorithm. Specifically, the basestation needs know about data that sensors have sampled and what their surrounding network topology looks like. To achieve this, sensors periodically transmit statistics in *summary messages* up the routing tree to the basestation. A summary message contains a coarse histogram over recent data, some network topology information, as well as the lowest, highest, and sum of all values over recent data, as well as the ID of the last complete storage index it has received from the basestation (see Section 5.3).

The basestation always saves the last histogram it receives from each node, thus allowing it to reason about a node even if newer summary messages are lost. In our experiments about 40% of summary messages do not reach the basestation, mostly due to network congestion near the basestation. Consequently, the basestation may have old statistics for some nodes, but, in practice, this does not significantly impair the overall performance of a storage index.

Summary histogram: The histogram part of the summary message captures the distribution of sensor readings on that node over its recent history. It consists of $nBins$ fixed-width bins (in our implementation, $nBins$ is 10). The value in bin n is the number of readings between $min + n((max - min + 1)/nBins)$ and $min + (n + 1)((max - min + 1)/nBins)$, where min and max are the smallest and largest values the attribute has taken on at s during recent history. For example, if $min = 1$, $max = 100$, and $nBins = 10$ and a node produced 8 readings between 50 and 60, the value of the 6th bin ($n = 5$) in the histogram would be 8.

A node needs its own recent readings to build this histogram and, therefore, writes its own readings in round-robin fashion to a fixed-size *recent-readings buffer* (size 30, in our experiments). This ensures that summary messages always contain histograms over the node's most recent data.

For the basestation to compute $P(p \to v)$, i.e., the probability that, in the future, a certain node, p, will produce a certain value v, p's histogram is used as follows (assuming that the probability that a sensor takes on any value in a bin is uniformly distributed):

$P(p \to v)$ {
  $binWidth = (max - min + 1)/nBins$
  $bin = (v - min)/binWidth$
  $P(v|bin) = 1/binWidth$
  $P(bin) = height(bin)/(\sum_{b\in \mathrm{Bins}} height(b))$
  return $P(v|bin) \cdot P(bin)$
}

Summary topology info: The topology part of the summary message contains a list of the node's n best connected neighbors (12, in our experiments), sorted by link-quality. A neighbor may or may not be a parent or child in the routing tree. (A node establishes link-quality from its neighbors by snooping the network and, per neighbor, counting the number of packets it did not receive using a monotonically increasing number that all nodes put in the header of all their outgoing packets.)

In addition to learning about nodes' neighbors this way, the basestation also learns about parent/child relationships in the routing tree through Scoop's custom packet header: each packet specifies the packet's origin and the origin's parent. Network neighborhood information from summary packets and the routing tree information from Scoop's packet headers allow the basestation to estimate the expected number of transmissions (xmits($x \to y$) in Figure 2) between any two nodes.

5.3 Mapping messages

After generating a storage index (see Section 4), the basestation splits it into different *mapping messages* since it is unlikely to fit in a single network packet. Scoop uses Trickle [13] to disseminate these storage index "chunks" to all nodes. Trickle uses a gossip-based probabilistic flooding protocol to disseminate data throughout a sensor network. To reduce communication overhead, the storage index is compacted by coalescing consecutive values that map to the same node into a single value range to node mapping. When a node has received all chunks for one storage index, it starts using that storage index, discarding the older index. Nodes do not synchronize this transition with other nodes.

Unfortunately, mapping packets may get lost, leaving nodes with incomplete storage indices. In that case, nodes continue to use the older complete storage index they have. This allows the basestation to avoid communication overhead by suppressing the dissemination of a new storage index altogether if it is very similar to the previous storage index; nodes will simply continue to use an older storage index. It also allows the basestation to more easily determine which nodes to query at query time—something that would be unduly complicated if nodes were to use half-assembled storage indices (see Section 5.5). If a node has never received a complete storage index, it stores all its data locally. The next section discusses how to route data between nodes who may be using different storage indices.

5.4 Routing sensor data

When a node produces a data item, it looks up the value's owner in its local copy of the storage index and sends (if the node itself is not the value's owner) a *data message* to the owner telling it to store the data. This section explains how data messages are routed in Scoop. Though this routing algorithm does not always allow any node to contact any other node in the network, it is very simple, requires relatively little network state, and works quite well in our implementation.

The goal of Scoop's routing algorithm is to route a certain value, v, to its owner, o, as dictated by the latest storage index, even if the node that produced v does not have the latest storage index. To achieve this, a data message contains three fields: the data item itself (v), an owner node (o), and a storage index ID (sid), all three of which are initialized by v's producer, i.e., the node that initiates routing. However, o and sid may be overwritten by nodes with a newer storage index, i.e., a storage index with a higher ID than sid. On receiving or producing a data item, a node n applies the following routing rules (in order):

1. If n's storage index is newer than sid, look up v in n's storage index and update o and sid in the packet header.

2. If $o == n$, store data locally on n: write data to the circular *data buffer*.

3. If o is in n's neighbor list, send the packet directly to that neighbor, irrespective of the routing tree.

4. If n is the base station, store it locally, i.e., don't route packets down the tree again.

5. If o is a node in n's descendants list, send the packet down the appropriate child branch.

6. Otherwise, send data item to n's parent.

Step 1 allows nodes with storage index newer than sid to modify the destination of the packet. Step 2 states that the packet has reached its destination. (Notice that the *data buffer* is separate from the *recent readings buffer* mentioned in Section 5.2.) Step 3 is an optimization that uses the neighbor list to take shortcuts through the routing tree. Step 4 is an optimization that prevents packets from being routed needlessly once they have reached the basestation. Step 5 sends the packet towards one of the node's descendants, if the destination is in the descendants list. In step 6, a node sends the packet to its parent—this step may be invoked repeatedly until a packet reaches the basestation.

Step 5 relies on a node's descendants list, which, as mentioned in Section 5.4, has a limited size. If a packet is destined for a node that is n's child, but n does not have this destination in its descendants list, the packet will either end up going to the basestation through (multiple) invocation(s) of step 6 or will be routed through an alternate path, by virtue of steps 3 and 5.

As an optimization, Scoop reduces the number of data packets by batching up to n sensor readings destined for the same node together into one packet (by default we use $n = 5$). As soon as a node produces data for another node or the number of batched readings exceeds n, the message is sent.

As we describe in our experiments, we have used this routing strategy extensively in both simulation (on networks up to 100 nodes) and on a real world, 62-node sensor network. Our results show that about 85% of the time, the appropriate destination node is found to store a particular data value; the remaining 15% of the time, the value ends up being stored at the root of the network, because the destination node could not be found. As we describe below, these results are sufficient to allow us to sustain an aggregate storage rate of about 4 times what a simple send-to-base strategy can provide, and also substantially reduces the communications burden on the root of the network.

5.5 Queries

A user issues queries from the basestation. A query consists of a *select list* of attributes (e.g. light, temperature), a *time range* specifying a minimum and maximum timestamp of interest, and a set of *value ranges* specifying the minimum and maximum ranges of interest for each of the attributes. With a megabyte of Flash memory, a Scoop node can store about 670,000 12-bit sensor readings. Thus, at 10 Hz, users will be able to query about 1,000 minutes of historical data.

The basestation determines the set of nodes to be contacted for this query by consulting the storage index(es) for the specified attribute(s) and time-range(s). (Unlike nodes, the basestation never discards old storage indices.) The value ranges in the query are used to find the appropriate entries in storage indices that could have been active at the time specified in the query. This yields the IDs of one or more nodes to be queried. Since different storage indices (see Section 5.3) may have been active at the query time on different nodes, a particular value may be stored at different network locations, rather than just one. For that reason, the basestation examines all storage indices active at that time (specified in its collection of summary messages — see Section 5.2) to establish the overlapping set of all possible nodes that may have the queried values corresponding to the time in the query. Alternatively, a user can query values from one or more specific nodes, in which case the query just specifies a time range and the list of nodes.

Once it has established which nodes it needs to contact, the basestation encodes the query in a query packet and specifies which nodes it wants an answer from using a bitmap in the packet's header. (This puts an upper bound to the size of the sensor network; 128 nodes in our current implementation.)

Scoop uses a modified version of Trickle [13] to disseminate query packets: our version uses both the packet's bitmap and a node's neighbor and descendants list to selectively re-broadcast query packets. If a node's ID corresponds to a 1 bit in the bitmap, the node linearly scans its *data buffer* for matching tuples. (Given the current limited size of the buffer, a linear scan poses no significant overhead. An index may be necessary if the size of the buffer increases.) The node then sends a reply—even if no tuples matched the query—through the routing tree back to the basestation. In practice, it takes several seconds for the first replies to come back to the basestation. In the worst case, all nodes are involved in answering a query (if the user queries the entire attribute's domain), but in all other cases a (much) smaller subset of nodes is queried, because of Scoop's index of value ranges to single nodes.

As an optimization, the basestation may use data from its summary messages to answer queries, which requires no network traffic at all. For example, since summary message contain the maximum attribute value measured per time period, queries that ask for the maximum value can be easily satisfied. To answer historical queries in similar fashion, the basestation never discards any summary message.

For each query it issues, the basestation updates its statistics that keep track of the query rate, and which attributes and what value ranges get queried. These numbers are used to estimate P(user queries v) and the query rate used in the algorithm from Figure 2.

6 Experiments

We implemented Scoop in TinyOS [8] and ran it in simulation (using the TOSSIM packet-level network simulator [11]) and on a 62-node indoor testbed consisting of Mica2 and Cricket [4] motes. Because the Scoop basestation requires more memory and CPU power than current mote hardware can provide, we ran the basestation on a PC connected to a mote using EmTOS [6].

As shown below, results obtained from simulation experiments and experiments on the real testbed are similar (modulo topology differences), which we take to indicate that the experiments run solely in simulation are a good predictor of real-world performance.

As we argued before, energy consumption is dominated by communication overhead. Therefore, our cost metric is the total number of messages the nodes collectively send. The goal of our experiments is to compare Scoop against other storage policies using this cost metric under different loads. The systems involved in our experiments are:

name	description	implemented
SCOOP	hybrid storage policy	yes
LOCAL	store locally, broadcast queries	yes
BASE	send all data to basestation	yes
HASH	static hash to route data	no, analytically

SCOOP is an implementation of the system we describe in this paper, with one important change: the optimization described in Section 4 where Scoop can default to "store-local" (aka LOCAL) has been disabled. In LOCAL, nodes store all data locally and queries are flooded to all nodes in the network. In BASE, all nodes send their data up the routing tree to the basestation and queries have no associated cost. Assuming nodes are uniformly distributed, we expect, on average, each data item to be sent roughly halfway across the network. In HASH, a uniform, static hash function maps each value to a node in the network where it is stored. Particular data values can then be found by applying this hash function to find the desired nodes. This approach is similar to the proposal for geographic hash tables (GHTs) [21] described in the literature as in-network storage technique. We expect the overall storage costs of HASH to be comparable to the storage costs of BASE because, on average, each packet has to be sent roughly halfway across the network as a result of hashing. However, HASH will incur additional costs for querying. As noted in papers that originally proposed the HASH scheme [21], one major benefit of HASH over BASE is reduction in load on certain hotspot nodes, since during storage no node has to transmit more data than any other. We briefly evaluate Scoop according to this same metric below. Because we did not have a working implementation of HASH (in particular, we didn't have a routing protocol that could reliably route from any node to any other node) we evaluate the cost of this HASH approach analytically.

name	description	sim/testbed
REAL	trace of real light data	sim only
UNIQUE	produce value equal to node ID	both
EQUAL	all nodes produce same value	both
RANDOM	random values	both
GAUSSIAN	values distributed around mean	both

Since Scoop is sensitive to the actual data distribution, we generate sensor data according to different methods, enumerated in the table above. For REAL, we use a trace of light data collected from a 50-node indoor sensor network deployment [1]. Each time a node in our experiments needs to produce a value, it reads the next number from this trace and produces that. Because these sensors were deployed in the same building, their light readings are highly correlated. However, since TinyOS has no file system support, we could only use the REAL data trace in simulation. For RANDOM, nodes produce random numbers in the range [0,100]. For EQUAL, all sensors in the network produce the same value for the duration of the experiment. For GAUSSIAN, each sensor i randomly selects a mean value μ_i from the range [0,100], which it uses for the duration of the experiment. It generates readings by sampling from a uni-dimensional Gaussian with mean μ and variance of 10. This is meant to approximate the behavior of

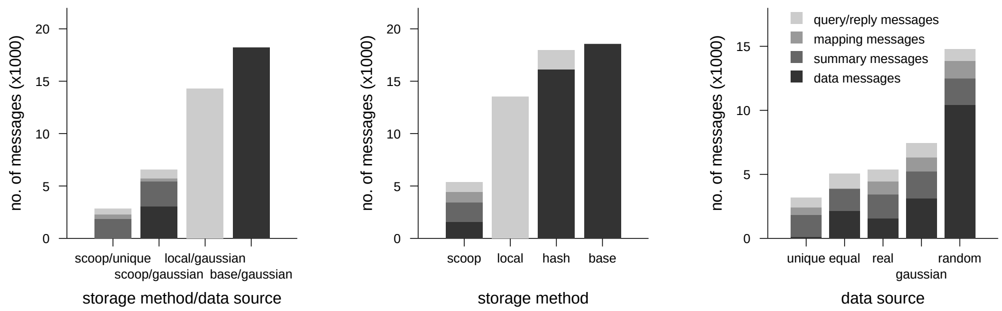

Figure 3. Left: Scoop compared to BASE and LOCAL on the testbed. Middle: Simulation results of Scoop compared to LOCAL, HASH, and BASE over the REAL data trace. Right: Simulation results of Scoop over different data sources.

a number of independent sensors generating data. For UNIQUE, each sensor produces its own, unique node ID as its value for the duration of the experiment.

The parameters we used in our experiment are listed below. All experiments use these default parameters, unless specified otherwise. All results we present are averages over three trials.

parameter	**value**	**remark**
attributes	1	
sample rate	1 in 15 seconds	
query rate	1 in 15 seconds	
summary rate	1 in 110 seconds	Scoop only
remap rate	1 in 240 seconds	Scoop only
size	62 nodes + 1 base	
duration	40 minutes	
data source	REAL	

By default, nodes sample their sensor (we measure only one attribute) once every 15 seconds. The basestation issues a query once every 15 seconds over 1-5% of the attribute's value domain (the query width). Nodes send a summary packet every 110 seconds and the basestation creates a new storage index ("remap rate") every 240 seconds which were values that worked well across a range of experiments. We experimentally vary the query width and data production rates in the experiments below.

The 62-node testbed is spread out across one floor of a large office building. The simulated topology also consisted of 62 nodes that, on average, can communicate with 20% of the nodes in the network at any given time, and of the pairs that can hear each other loss rates vary from twenty-five percent to about ninety percent. Connections are slightly asymmetric, as in most real wireless networks. All experiments ran for 40 (simulated) minutes. The first 10 minutes are spent stabilizing the network: nodes send heartbeat messages to form the routing tree. After the initialization period, nodes start sampling their sensor. Prior receiving their first storage index, nodes default to LOCAL storage.

Comparison of Scoop to other methods: Figure 3 (left) shows, per storage method, the breakdown of cost into data, summary, mapping, and query/reply messages on our mote testbed. Scoop running with UNIQUE performs very well on our testbed—each node produces its own, unique sensor reading, which allows Scoop to generate an optimal storage index. On the GAUSSIAN data source, Scoop outperforms LOCAL and BASE. In the BASE case, the only packets are data packets (from sensors to the basestation). In the LOCAL case, the only packets are query packets flooded to all nodes from the basestation and the resulting reply packets. SCOOP, with GAUSSIAN, adds some overhead for summary and mapping messages but, in doing so, finds an efficient storage index that vastly reduces the number of data, query, and reply packets. Note that we do not show HASH here because we can only evaluate it using an analytical model in our simulator.

Similarly, Figure 3 (middle) shows simulation results for different storage policies over the REAL data trace (in simulation). These results are similar to Figure 3 (left): Scoop adds overhead for summary and mapping packets for the storage index, but reduces overhead of other packet types. Note that HASH is included here, and, as expected, performs about as well as BASE since the query and data production rate are approximately the same.

Figure 3 (right) shows Scoop's performance over different data sources in our simulation. Scoop performs very well over UNIQUE since it exploits data locality. In RANDOM, however, there is no data locality at all for Scoop to exploit and so it performs no better than BASE or HASH. In EQUAL all nodes produce the exact same value; it incurs very few mapping messages because the basestation suppresses new mappings that do not change over time. EQUAL outperforms RANDOM even though every value has to be transmitted to a random node in both cases. EQUAL allows nodes to batch equal values (up to 5 in our experiments) before sending a data packet as described in 5.4. Note that RANDOM represents the case where there is no predictability in the data (e.g., past is not a good indicator of future values) and that in this case the system basically degenerates into performance that is equivalent to BASE or HASH.

The "unique" and "gaussian" columns, when compared to the "scoop/unique" and "scoop/gaussian" columns in Figure 3 (left), show that the relative performance of the simulation and real network are about

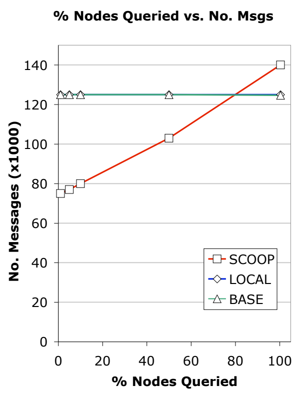

Figure 4. Cost as a function of percentage of nodes queried for different storage methods in simulation with REAL data.

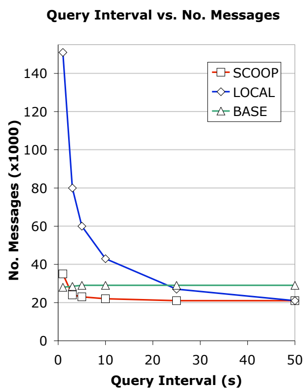

Figure 5. Total cost for different storage methods as a function of the interval between queries in simulation with REAL data.

the same, although the overall breakdown of messages is somewhat different due to variations in the topology used in the two cases.

Varying query selectivity: Figure 4 shows the cost of all storage methods with variable percentage of nodes queried. Note that LOCAL and BASE perform similarly because the sample rate and query rate are equal. LOCAL performs slightly worse due to the overhead of query dissemination, but this is negligible due to the efficiency of Trickle at disseminating queries.

LOCAL is unaffected by the percentage of nodes queried since it has to always query all nodes. Scoop outperforms both BASE and LOCAL for various query widths, but, around 60%, becomes slightly more expensive than BASE due to messaging related overheads. Clearly, when queries repeatedly ask for a substantial subset of the data (e.g., when the percentage of nodes queried is 100%), the best approach will be to simply ship the data out of the network (i.e., use BASE).

Varying query rate: Figure 5 shows the total cost for different storage methods as the query rate goes down, i.e., query interval goes up. Since the query cost is very small in SCOOP and zero in BASE, only LOCAL is substantially affected by this; as the query rate drops, it becomes a more attractive option relative to the others.

Other experiments: We measured the cost of Scoop running on different data sources as the sample interval increases (i.e., the rate at which data is stored decreases). As less data is stored, differences between the behavior of Scoop on different types of data are less pronounced as the cost of queries, mappings, and summaries becomes dominant.

In another experiment, we measured the loss rates of Scoop on the testbed. Data messages are successfully stored about 93% of the time, and about 78% of query results are successfully retrieved on average. This query success rate is comparable to the success rates obtained from other systems like TinyDB [17].

We also measured the number of messages sent by the root node in the network in the SCOOP, BASE, and LOCAL cases to compare the skew in the total number of transmissions. In most cases, the root node was the most active node. In the case of the FILE workload running in simulation, the root node in SCOOP sent about 4,000 mapping and query messages, and receives approximately 8,000 summary messages and 2,000 query reply messages. In the case of BASE, the root node receives about 24,000 data messages (and does no transmission). In the case of LOCAL, the root node send about 2,000 query messages and receives about 1,800 query reply messages. Hence, LOCAL places a lower burden on the root, but requires all nodes to receive and retransmit all 2,000 query messages (explaining its high overall cost), while BASE requires the root to do a great deal of reception (which is costly as the radio must be on at all times.)

This suggests that SCOOP has slightly more skew than LOCAL, since the root and nodes near it must consume additional energy due to handling of summary and mapping messages, but we believe this tradeoff is worthwhile as it allows SCOOP to use significantly less energy overall. In terms of overall energy consumption, this means that, for example, if a node running LOCAL can last for one month using a small battery, an average SCOOP node would last for about three months, although the battery on the root in SCOOP would have to be replaced every two weeks.

We also ran several experiments on different sized topologies (up to 100 nodes) in simulation, though we omit a detailed study of those results due to space constraints. We found that the system scaled well up to 100 nodes with little overall effect on loss rate. We observed that Scoop over a RANDOM distribution is more sensitive to larger networks as data is sent further across the network; Scoop over other distributions is less sensitive to network size.

We believe these real-world results demonstrate the practicality of Scoop —it runs on a large mote testbed, providing good overall performance using standard TinyOS networking protocols.

7 Related work

Ratnasamy et al. [21] compare the performance of a hashing-based approach called “data centric storage” with the performance of a local storage approach and a “ship-to-root” approach similar to our local storage and base storage methods described in Section 4. They show that hashing performs better in sensor networks that (a) are large and (b) collect data at high rates, but with an overall lower query rate. The overall performance of their approach is similar to that of the hashing scheme we compare against: it works well when the query rate is high relative to the data rate, but as the data rate gets high, the cost of routing data to a random location dominates the overall cost. Unlike Scoop, such in-network storage schemes are non-adaptive, choosing a random location for each value or event type according to a fixed hash function.

Liu et al. [15] propose a system that investigates the trade-offs between push and pull in query systems; these two opposites are analogous to our BASE and LOCAL schemes; as we show, the Scoop approach outperforms either of these approaches.

Li et al. [14, 7] propose a hash-based approach called DIM that strives to hash nearby sensor readings to the same node. This approach is well suited to range queries in sensor networks. Although the DIM approach is good for range queries, it suffers from the same limitations as GHT since it high data-storage cost because readings are sent far across the network and is non-adaptive.

Trigoni et al. [22] present a system that uses statistics about query frequency and data production rates to optimize network bandwidth in a multi-query environment. Their idea is to "push" data some distance up the network, towards then sink, and then "pull" the data the rest of the way when queries arrive. They tune the distance that data is pushed in the initial phase based on expected rates of querying and data production. Unlike our approach, they do not take into account the values that sensor produce or that queries ask for in determining how far to push data or where to store it. Kapadia and Krishnamachari [10] present a theoretical analysis of several such push-pull strategies, but also do not use a statistics driven approach.

There has been much work on building summaries and histograms in the database community that could be adapted to Scoop. Mannino et al. [19] summarize much of the early work in this area; our statistics are currently based on equal-bin-width histograms, and could benefit from more sophisticated summarization techniques.

8 Conclusion

By collecting statistics about network conditions and data and query rates in a store-and-query sensor network, Scoop periodically creates a storage policy that optimizes where sensors should store their data such to minimize overall communication. Scoop is a hybrid between several existing in-network storage approaches, sometimes acting like a purely local store when query rates are low and sometimes degenerating to the case where all data simply routed to the root of the network when query rates are very high. For this reason, Scoop almost always performs as well as, and usually much better, than existing approaches. Furthermore, our networking protocols work well despite high loss rates in sensor networks and we do not rely on complete network topology information or geographic routing protocols. Our results demonstrate that Scoop runs quite well on current generation medium-scale mote-based networks on the order of 100 nodes. For these reasons, Scoop is a core piece of our work on sensor network querying that we view as essential technology for future WSN-based monitoring deployments.

References

[1] Intel lab data. Web Page. `http://db.lcs.mit.edu/labdata/labdata.html`.

[2] R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, L. Krishnamurthy, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from the north sea and a semiconductor plant. In *Proceedings of SenSys*, 2005.

[3] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of MobiCom*, 2003.

[4] I. Crossbow. Wireless sensor networks (mica motes). `http://www.xbow.com/Products/Wireless_Sensor_Networks.htm`.

[5] A. Desphande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[6] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment heterogeneous sensor networks. In *Proceedings of SenSys*, 2004.

[7] R. Gummadi, X. Li, R. Govindan, C. Shahabi, and W. Hong. Energy-efficient data organization and query processing in sensor networks. In *ICDE*, 2005.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[9] A. Jain, E. Change, and Y.-F. Wang. Adaptive stream resource management using kalman filters. In *Proceedings of SIGMOD*, 2004.

[10] S. Kapadia and B. Krishnamachari. Comparative analysis of push-pull query strategies for wireless sensor networks. In *DCOSS*, 2006.

[11] P. Levis. Tossim: Accurate and scalable simulation of entire tinyos applications. `http://citeseer.ist.psu.edu/651380.html`.

[12] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of USENIX NSDI*, 2004.

[13] P. Levis, N. Patel, D. Culler, and S. Shekner. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of NSDI*, 2004.

[14] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *SenSys*, 2003.

[15] X. Liu, Q. Huang, and Y. Zhanh. Combs, needles, haystacks: Balancing push and pull for discovery in large-scale sensor networks. In *Proceedings of SenSys*, 2004.

[16] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of SIGMOD*, 2003.

[17] S. Madden, W. Hong, J. M. Hellerstein, and M. Franklin. TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb`.

[18] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*, 2002.

[19] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

[20] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51 – 58, May 2000.

[21] 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 *WSNA*, 2002.

[22] A. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Hybrid push-pull query processing for sensor networks. In *Proceedings of the GI Workshop on Sensor Networks*, 2004.

[23] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *ACM SenSys*, 2003.

[24] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.