Scoop: a hybrid, adaptive storage policy for sensor networks

Thomer M. Gil and Samuel Madden
CSAIL, Massachusetts Institute of Technology
Email: {thomer,madden}@csail.mit.edu

***Abstract*— One problem with existing store-and-query sensor networks is that they fail to take data and query rates or network topology information into account. This leads to expensive (and avoidable) communication overhead that reduces the lifespan of battery-powered sensor networks. Scoop reduces this overhead (up to a factor of four in our experiments) by dynamically creating and adapting an in-network storage policy based on statistics it collects about data, queries, and network conditions. Whereas existing in-network storage techniques are often at the extreme ends of the storage policy spectrum (e.g., store all data externally on a basestation, or store all data locally), Scoop's storage policy allows it to adapt between these extremes depending on the situation. The intuition behind our scheme is that data should be stored close to where it is needed, if it is needed. If query rates are high relative to the rate of data production, Scoop adapts its storage policy to store data closer to the source of queries. If query rates are low, the policy adapts to store data closer to its source. We have built a complete implementation of Scoop for TinyOS motes [3] and evaluated its performance on a 62-node testbed and in the TinyOS simulator, TOSSIM. Our results show that Scoop not only provides substantial performance benefits over alternative approaches on a range of data sets, but is also able to efficiently adapt to changes in the distribution and rates of data and queries.**

I. Introduction

Most existing approaches for data storage in store-and-query sensor networks can be broadly classified into one of two categories. One group of systems (e.g., Cougar [23] and TinyDB [15]) stores all data externally on a basestation; at the other extreme, systems store all data in-network (i.e., on sensor nodes), either by using some static hash-like function to map and send data to some remote sensor [19], [13], or by simply storing it locally, on the sensor where the data was collected. These existing storage approaches are typically tuned to work best for a particular query workload and data rate. For example, existing in-network approaches only work well when a small fraction of the total data items are ever accessed by a query; when every data item is queried, an external storage approach works better. Furthermore, none of these approaches account for variations in query rates or network conditions, leading to wasted communication that dramatically reduces the lifespan of battery-powered sensor networks.

In contrast, our system, Scoop, dynamically adapts its storage policy based on statistics about data, queries, and network conditions. The intuition behind our scheme is that data should be stored close to where it is needed, if it is needed. If query rates are high relative to data rates, Scoop adapts its storage policy to store data closer to the source of queries. If query rates are low, Scoop stores data closer to its source. That way, Scoop chooses the best policy for the current query workload and data distribution, allowing it to transition between the extremes of purely external or purely local storage.

Scoop expresses the desired storage policy in a *storage assignment* that is distributed throughout the network and tells each sensor where to store its data. A storage assignment can be thought of as a (periodically changing) map from values to a network location. 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. The basestation periodically recreates and redistributes this storage assignment based on various statistics it collects. By adjusting the storage assignment, Scoop can reduce the overall communication overhead by a factor of four (in our experiments) compared to existing techniques, despite the additional overhead created by distributing the storage assignment.

The usage model we envision is one where a user deploys a sensor network to collect one or more numerical attributes (such as temperature, humidity, light, or vibration) of the environment and then occasionally sits down at a basestation (a regular PC) connected to the sensor network to issue queries (in this work we focus on one-shot queries, but our techniques generalize to standing queries as well). The system does not know at deployment-time when or at what interval the user will do this, what her queries will look like, and at what rate she will issue these queries. Some example queries that we might expect to see are "find sensors with a temperature in range 20-40 degrees" or "find sensor with highest humidity during time T1-T2". In response to queries, the basestation uses the storage assignment as an index to identify nodes with relevant readings, sends them the query, and passes the reply to the user.

There are a number of applications where our techniques are relevant. Consider, for example, a sensornet deployed for monitoring a factory floor that uses sensors on equipment to measure, for example, 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.) Though current deployments (like [2]) typically send all sensor readings to a centralized basestation for analysis, 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 would then query the history of readings relevant to their interests. Different users might query for different types of readings: a maintenance worker may be interested in recent problematic conditions (e.g., temperatures or vibrational energy over some threshold), whereas a foreman or line manager may be interested in a longer-term history of machine temperature profiles or power consumption. Depending on the application and the rates of data production and querying, users may query for most or all of the values over time, or they may query for only a small subset of the total readings that are detected and stored. Scoop is designed to perform well on either workload.

We have built a complete implementation of Scoop for TinyOS motes [3] and evaluated its performance on a 62-node testbed and in the TinyOS simulator, TOSSIM. Our results show that Scoop not only provides substantial performance benefits over alternative approaches on a range of data sets, but is also able to efficiently adapt to changes in the distribution and rates of data and queries. In this paper, we focus on our experiences building a usable system for this kind of a medium scale (order 100 node) mote-based sensor network. Our approach is not designed to scale to very large networks of motes. Rather, we view heterogeneity and hierarchy as the way to scale up network size: we imagine assembling several medium-scale networks each running Scoop and connecting the basestations of these networks together through higher bandwidth networks (e.g., 802.11) rather than simply adding motes to our current network. This appears to be the approach taken by most large sensornet deployments [7], [2].

Scoop includes a number of features that are not present in existing sensor network query systems ([15], [23]). First, Scoop monitors and efficiently collects statistics about changes in the distribution and rates of data and queries, as well as changes in network conditions. Secondly, Scoop adaptively and strategically places data in the network (based on those statistics) making it very power efficient across a range of conditions. Finally, Scoop runs on current, low-power mote hardware, uses standard and well-understood networking protocols, and does not rely on hard-to-implement features such as localization, geographic routing, or precise time synchronization.

This paper describes our goals and challenges in Section II, explains desirable properties of a storage policy and outlines Scoop's algorithm to create one in Section III, and motivates various aspects of Scoop's design in Section IV.

II. Goals and Challenges

Scoop operates on a network of nodes that sample data at a certain *sample rate*. Periodically, the user issues queries over this data from a basestation at a certain *query rate*. Queries consist of a range of values or a list of nodes to be queried. In this work, we focus on one-shot queries over one or more indexed attributes.

Scoop is designed to work on current mote-class hardware; our implementation runs on Mica2 and Cricket nodes [3] and is written in TinyOS [9]. Hence, we assume an environment with limited power and radio bandwidth. We believe these resources will remain relatively scarce into the near future. 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.

Scoop, therefore, aims to minimize the communication overhead by storing data in-network. It handles changes in data rates, query rate, and network conditions by continuously collecting statistics from nodes and using these at the basestation to periodically create a *storage assignment*—a mapping from attribute values to network location. Rather than simple attribute values, a storage assignment can also map more complex values that are the outcome of some computation over one or more attributes, but in this paper we focus on building and maintaining a storage assignment over the values of a single attribute. (Like multiple indices in a database system, building multiple storage assignments for different sets of sensor readings or multi-dimensional storage assignments over multiple attributes is possible; see Section VI.)

Implementing this basic idea presents a number of challenges. First, the basestation needs to periodically create a storage assignment that strategically assigns data to certain network locations to minimize communication overhead (Section III). Secondly, Scoop needs to efficiently collect relevant statistics from nodes and send them to the basestation (Section IV-B) to be used as input to the algorithm that builds a storage assignment. Thirdly, Scoop needs to efficiently disseminate storage assignments to sensors (Section IV-C) and deal gracefully with the scenario when this (partially) fails due to packet loss. In addition, sensors need to route their data according to the storage assignment without keeping full routing state (Section IV-D). Finally, we need a query/reply mechanism on top of all this (Section IV-E).

III. Storage Assignment

This section motivates the design of Scoop's storage assignment. We first divide existing storage techniques for store-and-query sensor networks into three categories and briefly review each one.

One possible sensor network storage technique is "send-to-base": sensors send all their data to the basestation through a network routing tree rooted at the basestation. This is relatively simple to implement and queries can be satisfied at no cost (since all data is present at the basestation). Unfortunately, it can be wasteful: the system invests energy in 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. Note, however, that send-to-base is efficient when most data items are queried and the query rate is higher than data rates.

The second category 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.

A third category is a variation on "store-local": DHT-style systems that hash data to a certain location in the network. GHT [19], for example, stores data in the network by hashing data to a geographic location and routing the data to the nearest node. Queries for certain values can be similarly hashed and routed to the appropriate node(s). Queries are relatively efficient because they can be routed to just the relevant node(s), but, unfortunately, all data must be sent to a random destination that may be far from the node producing it. In particular, this approach will not perform well if more data is produced than queried. However, if queries eventually select most or all of the data (the only case where this approach appears attractive), then a "send-to-base" approach will be almost as good (modulo problems related to overloading nodes close to the basestation).

Send-to-base and store-local are two ends of the storage policy spectrum that Scoop spans to form an adaptive storage policy: 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, similar to DHT-style systems, but in Scoop the location is not determined by a static hash function but by a *storage assignment* that is periodically updated by the basestation and then broadcast to all nodes.

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

Fig. 1. A storage assignment for temperature.

A storage assignment is a value to node ID mapping. In this paper we simply map attributes ranges to node ID, as illustrated in Figure 1. One such mapping exists per attribute, per time period. The basestation creates a storage assignment based on statistics over the previous few minutes/seconds. (In our experiments, the basestation recreates a new storage assignment every 4 minutes). The mapping is chosen to minimize the total number of messages the system sends. 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 [10], [5].

Figure 1 shows a temperature storage assignment 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.

Clearly, the particular assignment of values to nodes 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 who produce this value to that 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 assignment is guided by the following design principles that a communication efficient storage assignment should follow: **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 assignment 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 assignment is outlined in Figure 2. (We focus on salient details of the algorithm rather than other, less important details of the real implementation.) 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 $\rightarrow$ node mappings is the storage assignment. (Section IV-B discusses how the basestation actually obtains these statistics.)

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

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.

Fig. 2. Storage assignment 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 assignment 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 IV-B.

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 most 150 (but usually much smaller) and n was 62. For the size of sensor networks we are aiming for (order 100), this is still 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 assignment and uses it if the expected cost is lower than the cost of the best storage assignment.

A possible 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 significantly reduce communication overhead if multiple regions in the network exhibit similar data distributions by assigning an owner per region. However, it may increase the size of a storage assignment 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.

IV. Scoop Design

In addition to **answering user queries**, a Scoop sensor network periodically **collects statistics**, shipping them to the basestation over a **routing tree**. The basestation periodically creates a new storage assignment and **disseminates** it throughout the network. Nodes then **route sensor data** between each other using this assignment. In this section we discuss the design of these parts of Scoop.

A. 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 IV-D) and routing queries (Section IV-E). 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 IV-D.)

B. Statistics Collection

The basestation relies on various statistics to run the storage assignment 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 so called *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 assignment it has received from the basestation (see Section IV-C).

Sending summary messages at a higher rate provides the basestation with more accurate data but incurs a higher cost. Conversely, lowering the summary message rate reduces the overhead, at the expense of inaccurate data at the basestation and, hence, a lower-quality storage assignment. This frequency is currently a tunable parameter, but a possible improvement is to dynamically adjust it.

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 assignment.

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 constructed as follows (assuming that the probability that a sensor takes on any value in a bin is uniformly distributed):

```
$P(p \to v)$ {
    $binWidth = (max - min + 1)/nBins$
    $bin = (v - min)/binWidth$
    $P(v|bin) = 1/binWidth$
    $P(bin) = height(bin)/(\sum_{b \in \text{Bins}} height(b))$
    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 custom packet headers allows the basestation to estimate the expected number of transmissions (xmits($x \to y$) in Figure 2) between any two nodes.

C. Mapping messages

After generating a storage assignment (see Section III), the basestation splits it into different *mapping messages* since it is unlikely to fit in a single network packet. Scoop uses Trickle [12] to disseminate these storage assignment "chunks" to all nodes. Trickle uses a gossip-based probabilistic flooding protocol to reliably disseminate data throughout a sensor network. To reduce communication overhead, the storage assignment is compacted by coalescing consecutive values that map to the same node into a single value range to node mapping.

Each mapping message contains a monotonically increasing ID to identify the storage assignment that the mapping message is part of and the total number of entries in the storage assignment so that receiving nodes know when they have received the entire storage assignment. The remaining space in the packet is filled with $(valuefrom - valueto, nodeID)$ tuples, i.e, the storage assignment "chunk". When a node has received all mapping messages for one storage assignment, it starts using that storage assignment, discarding the older assignment. Nodes do not synchronize this transition with other nodes.

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

D. Routing sensor data

When a node produces a data item, it looks up the value's owner in its local copy of the storage assignment 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 without requiring nodes to keep full routing state (which is possibly large and difficult to keep up-to-date).

The goal of Scoop's routing algorithm is to route a certain value, v, to its owner, o, as dictated by the latest storage assignment, even if the node that produced v does not have the latest storage assignment. To achieve this, a data message contains three fields: the data item itself (v), an owner node (o), and a storage assignment 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 assignment, i.e., a storage assignment 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 assignment is newer than sid, look up v in n's storage assignment 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 assignment 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 IV-B.) 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 IV-D, 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. In our experiments—a 62-node network with descendants list of size 32—we are not aware of running into this situation.

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.

E. 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. (Note that there is a limit to the amount of historical data a node can store in RAM: currently, the *data buffer* stores under 100 data items, but this can be easily expanded by using Flash memory, rather than the relatively small RAM on motes.)

The basestation determines the set of nodes to be contacted for this query by consulting the storage assignment(s) for the specified attribute(s) and time-range(s). (Unlike nodes, the basestation never discards old storage assignments.) The value ranges in the query are used to find the appropriate entries in the storage assignment, yielding the IDs of one or more nodes to be queried. (Since nodes may be using different storage assignments (see Section IV-C), values may end up being stored at different network locations, rather than just one. For that reason, the basestation uses the storage assignment ID specified in its collection of summary messages (see Section IV-B) to establish the overlapping set of all possible nodes that could be storing queried values.) 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 to be queried.

Once it established which nodes it has 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; 64 nodes in our current implementation.)

Scoop uses a modified version of Trickle [12] 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 assignment of value ranges to single nodes.

In our experiments, up to 80% of query packets reach the relevant nodes, but replies frequently get lost (only 30%-40% of replies reach the basestation), most likely due to congestion near the basestation. (We note that these loss rates are consistent with loss rates observed in other sensor network applications [20], [16].) A future improvement could be for the basestation to re-issue queries, specifying in the bitmap only those nodes from which it has not yet received a reply.

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.

F. Network failures

Scoop deals with network failures through a range of techniques. First, we use a limited number of link-level retransmissions for data, summary, query, and reply messages. Scoop uses Trickle [12] to ensure that all nodes receive the mapping and query messages. Also, nodes insert a random delay before sending mapping, summary, and reply messages to avoid congestion near the top of the routing tree. In addition, nodes will suppress sending a summary message if the difference with the last summary message is insignificant. Similarly, the basestation may suppress dissemination of a storage assignment if it the difference with the previous storage assignment is insignificant. Also, a node batches multiple data items into a single data message. Finally, the basestation uses old statistics when summary messages from a node are lost.

V. EXPERIMENTS

We implemented Scoop in TinyOS [8] and ran it in simulation (using the TOSSIM packet-level network simulator [18]) and on a 62-node indoor testbed consisting of Mica2 and Cricket [3] 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 [7].

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 III 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 hash function maps each value to a node in the network. We expect the storage costs of HASH to be comparable to the storage costs of BASE. Since, on average, each packet has to be sent roughly halfway across the network. However, HASH will incur additional costs for querying. Because routing to a random node from any node in the network requires a non-tree based routing algorithm—typically based on geographic routing (such as GPSR [11])—we can only measure the cost of HASH analytically, since we could not find a reliable implementation of such an algorithm and nodes in our network do not have access to geographic information.

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 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 assignment ("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 about half of the nodes in the network, 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 assignment, nodes default to LOCAL storage.

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 assignment. 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 assignment 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 assignment, 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. 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

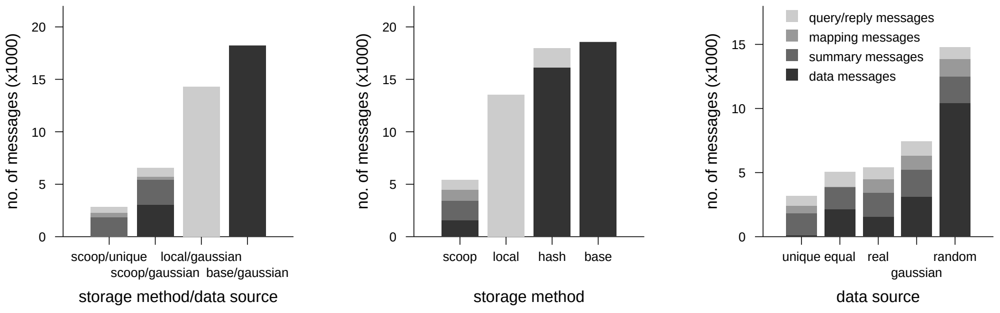

Fig. 3. **Left:** Scoop compared to BASE and LOCAL on the testbed. **Middle:** Simulation results of Scoop compared to LOCAL, HASH, and BASE over the REAL data trace. **Right:** Simulation results of Scoop over different data sources.

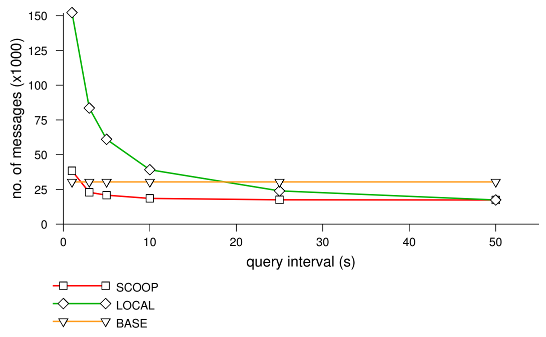

Fig. 4. Total cost for different storage methods as a function of the interval between queries.

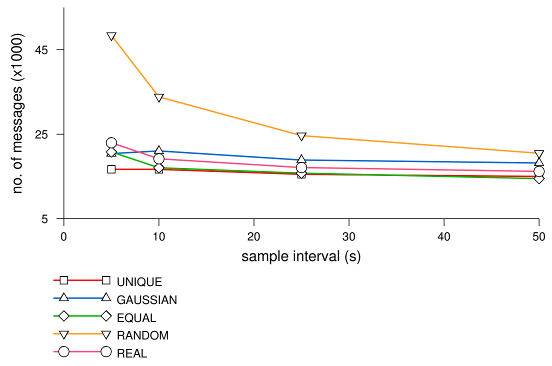

Fig. 5. Scoop cost as a function of sample interval for different data sources.

packet as described in IV-D. 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 the same, although the overall breakdown of messages is somewhat different due to variations in the topology used in the two cases.

Figure 4 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.

Figure 5 shows 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, the differences between the behavior of Scoop on the different types of data becomes less pronounced; the cost of queries, mappings, and summaries becomes dominant.

We measured Scoop's performance as a function of the number of nodes involved in replying to a query. (Recall that LOCAL contacts all nodes and BASE contacts no nodes to answer a query.) Scoop's query/reply overhead increases as more nodes are contacted, but continues to dominate LOCAL and BASE for queries that contact fewer than 70% of the nodes.

We also measured the loss rates of Scoop on the testbed. Data messages are successfully stored about 93% of the time, and about 39% of query results are successfully retrieved on average. This relatively low query success rate is due to the fact that we do not currently have any retries on query or replies.

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.

VI. Extensions and Future Work

The basic Scoop architecture makes it easy to support a number of extensions; each of these requires very few changes to the code, mostly to the storage assignment construction algorithm. **Multiple owners per value:** As mentioned in Section III, computing an owner *set* rather than a single owner per value could improve Scoop's performance when multiple regions in the network exhibit similar data distributions. **Range query optimizations:** optimizing the placement of sensor values that have a high probability of being queried together could improve the performance of range queries. **Multi-dimensional queries:** Currently, Scoop's storage assignment is for a single attribute; supporting multi-dimensional storage assignments could improve the performance of some queries. One approach would be to partition data according to multiple attributes and assign cubes in this multidimensional space to different nodes (e.g., node 1 would be owner for temperatures in the range 10–12 where light is in the range 100–120).

VII. Related work

In-network Storage in WSNs: Ratnasamy et al. [19] 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 III. 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. Scoop improves on GHT in two ways: (1) it eliminates the need for geographic routing which is difficult to implement and requires nodes to be location-aware, and (2) instead of hashing, Scoop strives to minimize the combined cost of querying and storing data based on current query rates and the values sensors have recently produced. As we illustrated, Scoop strictly dominates the performance of hashing-based schemes.

There has been other work in the WSN community on in-network storage. Ganesan et al. [6], [22] investigated wavelet-based schemes for summarizing data inside a sensor network; they envision nodes storing data locally and transmitting summaries of it out of the network. Their wavelet based techniques are complimentary to ours: wavelets could be a useful mechanism for building summary messages and approximation techniques are an interesting future direction for us to explore. Desnoyers et al. [4] built a two-tier system for data collection and storage in sensor networks. Our focus in this paper was on storage issues in a medium-sized homogeneous network; if we wished to scale to much larger networks, such two-tiered techniques would become important for us as well.

Liu et al. [14] 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. [13] 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: it requires geographic routing and has a high data-storage cost because readings are sent far across the network.

Trigoni et al. [21] 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.

There has been much work on building summaries and histograms in the database community that could be adapted to Scoop. Mannino et al. [17] 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.

VIII. 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 are robust to a range of failures that are common 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] I. Crossbow. Wireless sensor networks (mica motes). http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

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

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

[6] D. Ganesan, D. Estrin, and J. Heidemann. Dimensions: Why do we need a new data handling architecture for sensor networks? In *Proceedings of the First Workshop on Hot Topics In Networks (HotNets-I)*, Princeton, New Jersey, 2002.

[7] 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.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

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

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

[11] B. Karp and H. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, 2000.

[12] 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.

[13] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceeding of the First ACM Conference on Sensor Systems (SenSys)*, 2003.

[14] 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.

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

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

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

[18] T. A. Philip. Tossim: Accurate and scalable simulation of entire tinyos applications.

[19] 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 SensorNets, 2002.

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

[21] 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.

[22] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. B. R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Processings of SenSys*, 2004.

[23] Y. Yao and J. Gehrke. Query processing in sensor networks. In *Conference on Innovative Data Systems Research (CIDR)*, 2003.