POS: A Practical Order Statistics Service for Wireless Sensor Networks

Landon P. Cox
EECS Department
University of Michigan
Ann Arbor, MI, USA
Email: lpcox@eecs.umich.edu

Miguel Castro
Microsoft Research
Cambridge
UK
Email: mcastro@microsoft.com

Antony Rowstron
Microsoft Research
Cambridge
UK
Email: antr@microsoft.com

***Abstract*— Wireless sensor networks are being deployed to monitor a wide range of environments. Since energy efficiency is critical in these deployments, previous work proposed in-network aggregation and approximation techniques to reduce the energy consumed collecting data. In-network aggregation can be used to compute statistics such as max, min, and average accurately and energy-efficiently, but it does not work well for order statistics such as median. Current approximation techniques to compute order statistics cannot deliver both good accuracy and energy efficiency. This is unfortunate because order statistics are more resilient to faulty sensor readings than max, min, or average.**

We present the design and implementation of POS, an in-network service that computes accurate order statistics energy-efficiently. POS returns a stream of periodic samples from any order statistic. It initially computes the value of the order statistic and then periodically runs a validation protocol to determine whether the value is still valid. If not, it uses an optimized binary search to determine the new value and then resumes periodic validation. POS uses in-network aggregation and transmission suppression to reduce communication complexity. Results from both experiments on a mote testbed and simulations show that POS can compute order statistics accurately while consuming less energy than the best techniques to compute averages in common cases.

I. Introduction

Wireless sensor networks are currently used to monitor a number of different environments, including the Golden Gate Bridge [1], Great Duck Island [2] and Tungurahua volcano [3]. Energy is the critical resource in sensor nodes because they are battery powered and replacing batteries in these environments is expensive, difficult, or even dangerous. Typically, wireless communication is the biggest drain on energy. Users query the sensor readings through a base station and the sensor nodes form a multi-hop wireless network connected to the base station. It is important to reduce the communication induced by user queries to conserve energy and extend the lifetime of the sensor nodes.

Previous work has developed techniques to compute certain types of queries accurately and with low communication complexity. For example, queries for max, min, and average sensor readings can be computed accurately and with low communication complexity.

However, previous techniques to compute order statistics (such as median) cannot provide both good accuracy and low communication complexity. This is unfortunate because order statistics can provide a more complete characterization of the distribution of sensor readings than max, min, or average. Additionally, they are more robust to outliers, which are common in sensor networks due to failures, poor calibration, or interference from the environment. For example, a single reading from a faulty sensor can significantly change the average reading value, but order statistics like the median, 95th percentile and 5th percentile are resilient to these failures. This paper describes techniques to compute order statistics accurately and with low communication complexity.

TAG [4] maintains a spanning tree rooted at the base station to execute user queries. It computes order statistics by sending all sensor readings to the base station. This technique is accurate (in the absence of mote failures and message losses), but it does not scale. It imposes a communication complexity of $O(N)$ on the sensor nodes near the base station, where N is the number of nodes in the network. These nodes are critical because when their batteries expire, users can no longer query the sensor network.

Other projects have studied approximation techniques for computing order statistics [5], [6]. These techniques cannot provide both good accuracy and energy efficiency. Recently, Patt-Shamir [7] introduced a technique to compute order statistics accurately by using in-network aggregation to compute the number of sensors with readings greater than a threshold. It performs a binary search by repeating the query with adjusted thresholds until the threshold equals the desired order statistic. This technique achieves $O(1)$ communication complexity per sensor node but our results show that the constants involved are large; this technique requires more energy to compute a median than sending all the readings to the base station even for relatively large networks.

We propose POS, a practical service to compute order statistics. POS implements persistent queries [4]: it returns a stream of periodic samples from an order statistic specified by the user with a period specified by the user. POS starts by computing the initial value of the order statistic and then stores that value at all sensor nodes using a spanning tree rooted at the base station [4]. Periodically, POS runs a *validation* protocol to determine if the value is still valid. If not, it uses an *update* protocol to determine the new value and then resumes periodic validation. The update protocol uses an optimized

version of the binary search algorithm in [7]. POS can also answer top-k queries [8].

POS implements in-network aggregation and transmission suppression techniques that exploit the properties of order statistics to reduce communication complexity during both the validation and update protocols. Instead of counting the number of sensor readings greater than a threshold [7], POS counts the number of sensor readings that become larger and smaller than the previous value of the order statistic. These counts can be aggregated in the network and they enable very effective transmission suppression. For example, leaf sensors do not send messages if the order of their reading relative to the previous value of the order statistic did not change. Similarly, interior nodes in the tree can suppress transmission and the base station can avoid the update protocol if the number of movements of both types is equal. POS computes the exact value of the order statistic (in the absence of mote failures and message losses) and achieves $O(1)$ communication complexity per node with a small constant. Our mote implementation uses acks to mitigate the effects of message losses.

We implemented POS and evaluated its performance using both the TOSSIM simulator [9] and experiments on Harvard's MoteLab [10]. Our results show that POS can compute accurate order statistics with significantly lower communication complexity than the techniques used by Patt-Shamir [7] and TAG [4]. POS does particularly well in the common case when sensor reading change slowly relative to the period specified by the user. In this case, POS requires less energy to compute order statistics than TAG requires to compute averages.

The rest of the paper is organized as follows. Section II lays out the basic communication model used by POS. Section III describes the initialization, validation, and update protocols POS uses to maintain persistent order statistics. Section IV describes and evaluates our POS prototype. Finally, Section V describes related work and Section VI contains our conclusions.

II. Background

Typically, sensor networks are connected to the outside world through a base station. These networks use sensors such as the Crossbow motes [11] with a CPU, memory, battery, several sensors, and a low-power wireless radio. In most cases, the motes are distributed over an area large enough that most are unable to communicate directly with the base station. Therefore, they self-organize into a multi-hop, ad-hoc wireless network. Users interact with the sensor network through the base station: a query is sent to the base station, the base station communicates with the motes to compute a reply to the query, which is forwarded to the user. This section discusses the basic communication primitives that POS uses to broadcast messages from the base station to all motes and to collect data from all motes at the base station.

The base station broadcasts a message reliably to all sensors using controlled flooding. The base station starts by broadcasting the message to the motes in its radio range. After receiving a message, a mote queues the message and waits for some time t before broadcasting it to its neighbors. t is chosen randomly from the interval between zero and a forwarding window, w. While waiting, the mote counts the number of times it receives the enqueued message. Once the wait is over, the mote broadcasts the enqueued message to its neighbors and checks how many copies of the message it received while waiting. If a mote received the enqueued message an additional k times, it does not broadcast the message again. Otherwise, the mote broadcasts the message again. In our prototype implementation k is three. This is similar to one of the techniques proposed in Trickle [12].

Redundant broadcasts like this have two important properties. First, motes in dense clusters will normally broadcast only once because they are likely to overhear the message from k of their neighbors while waiting. This saves transmissions and energy. However, in less dense areas, where motes have fewer neighbors and individual links are more important, motes may have to rebroadcast their message. These rebroadcasts reduce the impact of temporary link failure and ultimately help ensure that the message reaches all motes.

POS uses a routing tree rooted at the base station to collect statistics from the sensor network and schedules communication using the same techniques as TAG [4]. Each mote in the tree maintains a pointer to its parent, the overall depth of the tree, and its own depth but motes do not know their children.

Data is collected in *epochs* that are divided up into d_{max} communication *slots*, where d_{max} is the overall depth of the tree. During an epoch data flows up the tree with only one level transmitting in each slot to reduce the chance of collisions and enable nodes to switch off their radios to save energy. A mote's depth in the tree determines both the slot in which it receives data and the slot in which it transmits data. Motes at depth d_{max} do not receive data from children and they transmit data in the slot number one in the epoch. Other motes switch their radios on to receive data in slot number $d_{max}-d$, aggregate the data, and then transmit the aggregated data to their parent in slot $d_{max}-d+1$, where d is their depth in the tree. Nodes pick a random instant in their slot to transmit data to their parent to avoid interference. The base station receives the aggregate values from its children in slot number d_{max}.

Each mote begins with a static value of d_{max}. When the network is initialized, the base station broadcasts a request for network statistics. This request returns a count of all motes as well as the maximum known depth. If a mote has a depth greater than its initial d_{max}, it participates in the first slot and sets the maximum depth appropriately. If the request returns a maximum depth that is larger than the initial d_{max}, the base station sets d_{max} to this value and broadcasts it to all motes.

III. Design

POS implements persistent queries that return a stream of periodic samples of an order statistic. The user specifies the order statistic such as median or 95th percentile, together with the period between samples, referred to as the *epoch*. We call these queries *persistent order statistics*.

POS implements persistent order statistics using three protocols; *initialization*, *validation*, and *update*. The initialization protocol computes the current value of the order statistic at the base station (using a technique similar to TAG [4]) and replicates this value at all motes. Every epoch, POS runs a validation protocol that uses these replicas and new readings taken by the motes to determine if the current value of the order statistic is still accurate. If the current value is accurate, it is returned to the user. Otherwise, POS runs the update protocol to compute the new value of the order statistic and then replicates this value at all motes.

Next, we briefly describe the state maintained in the base station and motes. The remainder of this section describes each protocol in detail.

A. State

POS maintains a small amount of state at the base station and motes in addition to the state required to maintain the routing tree. It maintains four values at the base station: the current estimate of the specified order statistic or percentile, p, and the number of motes whose readings are greater than p, less than p, and equal to p, which are denoted by g, l, and e respectively. For example when computing the median, p is a value for which both g and l are less than or equal to half the number of readings.

Each mote stores the latest value of the order statistic, p (broadcast by the base station) and the *order* of its previous reading relative to p. The order of a reading r can take three values that correspond to the three possible orderings of r relative to p, i.e., $r < p$, $r > p$ or $r = p$. The mote can store both p and the order of its previous reading in flash memory to survive reboots.

B. Initialization protocol

The initialization protocol computes the initial values of p, g, l, and e by asking all motes to forward their readings up the tree to the base station as in TAG [4]. The communication mechanisms described in the previous section are used to broadcast the request and collect the readings. The base station sorts the readings and computes the values. These values will be exact in the absence of mote failures and message losses.

Each mote sends its readings and the readings of their descendants to its parent using the least possible number of packets. However, the total number of bytes and the total number of packets still grows linearly with the number of motes. Motes that are leafs in the tree only send a packet with a single reading but nodes close to the root of the tree can send many packets with many readings. When this procedure is repeated often, this can result in short battery life for the nodes near the root of the tree, which limits the useful life of the entire sensor network. Since initialization is expensive, POS only does this once when the user first issues the query.

The initialization protocol replicates the value of the order statistic computed by the base station, p, at all motes using a broadcast. Motes then compute the order of the reading they sent to the base station relative to p and store p and the order.

O_{i-1}	O_i	$into_<$	$into_>$	$outof_<$	$outof_>$
$<$	$>$	0	1	1	0
$<$	=	0	0	1	0
$>$	$<$	1	0	0	1
$>$	=	0	0	0	1
=	$<$	1	0	0	0
=	$>$	0	1	0	0

Fig. 1. Order Changes and Report Values

C. Validation protocol

The validation protocol runs at the beginning of every epoch i after the initialization protocol. Each mote takes a new reading r_i and then checks if this reading can invalidate the values of p, g, l, and e stored by the base station.

The check is performed by computing the order O_i of r_i relative to p (which is replicated locally) and comparing the result with the order of the previous reading relative to p, O_{i-1}, which is also stored locally. If $O_i = O_{i-1}$, the mote does not need to report anything to the base station, which saves a message transmission. For example, if a new reading is less than p just as the previous reading was, this cannot possibly affect p, g, l, or e. For readings that change slowly relative to the epoch duration, this will be the common case and will remove many expensive transmissions.

If the order changes from the previous reading to the new one, the mote must report the change to the base station through its parent. The *reports* sent by motes have four integer fields that encode all possible order changes: $into_<, into_>, outof_<$, and $outof_>$. Figure 1 shows how these fields are computed for the different combinations of values of O_{i-1} and O_i. The fields are currently represented using 16 bits.

Sending reports back to the base station has a big advantage relative to sending back readings — reports can be easily aggregated up the tree such that each node transmits a small constant number of bits to its parent. Each node aggregates its report with the reports of its children by summing the values of the corresponding fields before forwarding the result to its parent.

Once the base station receives reports from all its children, it aggregates those reports and uses the result to update g, l, and e as follows:

$$g := g - outof_> + into_>$$

$$l := l - outof_< + into_<$$

$$e := n - g - l$$

where n is the number of motes in the network. We assume that the number of nodes in the network is computed periodically using the technique described in [4] to take into account mote failures. This is a lightweight operation that can be performed infrequently in most settings, e.g., every fifteen minutes or half hour.

After updating its state, the base station checks if g and l are correctly distributed. If they are, the previous value of the order statistic is still valid and can be returned back to the user. For example, if both g and l are less than or equal to $n/2$, the previous value of the order statistic is still a valid median. Otherwise, the base station initiates the update protocol.

POS performs even more aggressive transmission suppression by using the observation that if the order of one mote's reading changes from $<$ to $>$ and the order of another mote's reading changes from $>$ to $<$, the two changes cancel each other. This allows motes to filter reports as they move up the tree. Before sending the aggregated report to its parent, each mote subtracts the larger of $into_<$ and $outof_<$ from the smaller and does the same with $into_>$ and $outof_>$. If the result of both subtractions is zero, the mote does not transmit the report to its parent.

The validation protocol is efficient because it suppresses many message transmissions and triggers the update protocol only when necessary. Suppressing message transmissions reduces energy consumption, but it also reduces message loss due to radio interference. The fewer motes transmitting, the more likely those who are will succeed without having to fall back on retransmissions. The cost of the validation protocol is $O(1)$ per mote; each mote transmits at most one message.

Statistics such as min, max, and average can also be computed with a single message per mote. However, transmission suppression techniques to compute these statistics cannot be as effective (without loosing accuracy) as those we described for order statistics. When computing exact values for min, max, and average, leaf motes can suppress transmission of their readings only when their readings did not change since the last epoch. This makes computing min, max, and average more expensive than computing order statistics when the readings change slowly relative to the epoch specified by the user. The same resilience to noise that makes order statistics more robust to sensor failures and miscalibrated sensors than min, max, and average also enables more efficient computation of order statistics in the common case.

D. Update protocol

The update protocol is invoked when the validation protocol determines that the value of the order statistic stored at the base station does not reflect the current sensor readings. Upon completion, it returns a new value for the order statistic. We expect it to be invoked infrequently when readings change slowly relative to the epoch duration. Nonetheless, the update protocol is very efficient. It uses a binary search over the range of possible sensor readings to compute the new value of the order statistic.

The initial interval for the binary search is determined using the values of l, g, and e computed at the end of the validation protocol and the previous value of the order statistic p. For example, when computing the median, p is greater than the current median if $l > n/2$ and less than if $g > n/2$. If p is greater than the current value of the order statistic, the initial interval for the search is between p and the minimum possible value in the sensor range. Similarly, if p is less than the current value of the order statistic, the initial interval for the search is between p and the maximum possible value in the sensor range.

After computing the initial interval for the search, the base station sets p to the midpoint of the interval and broadcasts p to all the motes indicating that it is updating its value. When a mote receives this message, it computes the order of its current reading relative to the new value p. Then it compares this order with the order relative to the previous value of p, stores the new order value, and generates a report if there is a change (as in Figure 1). These reports are aggregated up the tree with transmission suppression exactly as in the validation protocol.

The base station uses the aggregated reports to update l, g, and e as discussed before. If l and g are distributed appropriately, the value of p is the current value of the order statistic and it can be returned back to the user. Otherwise, the base station halves the search interval and repeats the process. It uses the updated values of l and g to determine which half of the search interval contains the current value of the search statistic.

The epoch duration is chosen such that the validation protocol and the update protocol can run during the epoch duration before sensors take a new reading. This is important to ensure that the sensor readings do not change during the search process. Otherwise, POS could not guarantee convergence.

The number of iterations in the search process is bound by the number of bits in a sensor reading, b, in the worst case. Therefore, the number of messages sent and the energy consumed by all motes in the update protocol, including those closest to the base station, is $O(b)$. The update protocol scales very well because this bound is constant with the number of motes.

However, if sensor readings are 16 bits, the update protocol could require 16 rounds to update the value of the order statistic. This is less energy efficient than simply forwarding all values to the base station for networks as large as 100. Because of this, we developed an optimized update protocol that significantly reduces the number of iterations required to update the order statistic, even in relatively small networks.

E. An optimized update protocol

The optimized update protocol adds hints to reports in addition to the four movement counters. These hints are sensor readings that allow the base station to shrink the search intervals and reduce the number of iterations required to complete the search.

To understand what hints to use and how they can be aggregated up the tree, we start by analysing the effect of individual order changes in sensor readings on the value of the order statistic. Let U be the total number of "up" moves (i.e., order changes into $>$) that an omniscient observer would see and let $u_1, u_2, \ldots, u_U$ be the sequence of the sensor readings after those moves sorted in increasing order (as shown in Figure 2). Similarly, let D be the total number of "down"

moves (i.e., order changes out of $>$) and $d_1, d_2, \ldots, d_D$ the sequence of the sensor readings after those moves sorted in decreasing order (as shown in Figure 2). Let p be the value of the order statistic before the moves.

Moves in opposite directions cancel, for example, u_U cancels d_D, u_{U-1} cancels d_{D-1} and so on. Therefore, we can compute the minimum interval for the binary search as follows:

- if $U - D = k > 0$ then the new value for the order statistic is between p and u_k.
- if $D - U = k > 0$ then the new value for the order statistic is between d_k and p.

Figure 2 demonstrates how the movements in opposite directions cancel. The top line shows the distribution of sensor readings (grey circles) with the current median (p). The bottom line shows the new sensor values. If the new value differs from the old value, the old value is shown as a white circle and the new value is shown as a black circle. In this example, $U = 3$ and $D = 2$, so $k = 1$. Since d_2 cancels u_3 and d_1 cancels u_2, the new median lies in the range from p to $u_k = u_1$.

Given this observation, the most obvious approach to compute the interval for the binary search is to simply send all readings corresponding to order changes back to the base station. However, this is expensive and, in the worst case, it will induce a communication complexity of $O(N)$ on the motes near the base station. Instead, we can reduce the overhead to a small constant and obtain a conservative approximation. This approximation will compute an interval that is guaranteed to contain the order statistic, but that may be longer than the intervals determined by an omniscient observer.

One way to achieve this is to add two fields to the reports in addition to the four movement counters—a min and a max reading. When a mote generates a change report, both the max and min fields in the report are set to its latest reading. These reports are aggregated in the obvious way. For example, the max field in the aggregated report is set to the maximum value in the max fields of the reports being aggregated. When the base station computes the final report, it uses the updated values of l and g to determine whether to use the interval between p and the max field in the report or between the min field in the report and p for the search.

This simple approach works, but has the disadvantage that it is very conservative. We can do better by applying the omniscient observer's reasoning locally at every mote. Instead of adding two hints to reports, the improved approach adds a single hint containing a sensor reading. When a mote generates a change report, it sets the hint value to its latest reading. Motes aggregate hints using the pseudo code in Figure 3.

The invariant for this procedure is that the updated order statistic obtained by applying all the changes accounted for in a report must be between p and the report's hint. This is clearly true for reports generated by leaves in the tree. The code in Figure 3 preserves this invariant.

It first decides whether there are more moves up or down in the merged report at Line 14. If there are more moves up, it only picks hints that are greater than p because the new value

AggregateHints (Report $r1$, Report $r2$)

```
1:  lts1 ← 0, gts1 ← 0, lts2 ← 0, gts2 ← 0
2:  if r1.into_< > r1.outof_< then
3:     lts1 ← r1.into_< − r1.outof_<
4:  end if
5:  if r2.into_< > r2.outof_< then
6:     lts2 ← r2.into_< − r2.outof_<
7:  end if
8:  if r1.into_> > r1.outof_> then
9:     gts1 ← r1.into_> − r1.outof_>
10: end if
11: if r2.into_> > r2.outof_> then
12:    gts2 ← r2.into_> − r2.outof_>
13: end if
14: if (lts1 + lts2) < (gts1 + gts2) then
15:    if (gts1 ≠ 0) AND (gts2 = 0) then
16:       return r1.hint
17:    else if (gts2 ≠ 0) AND (gts1 = 0) then
18:       return r2.hint
19:    else if (gts1 + gts2) − (lts1 + lts2) = 1 then
20:       return min (r1.hint, r2.hint)
21:    else
22:       return max (r1.hint, r2.hint)
23:    end if
24: else
25:    if (lts1 ≠ 0) AND (lts2 = 0) then
26:       return r1.hint
27:    else if (lts2 ≠ 0) AND (lts1 = 0) then
28:       return r2.hint
29:    else if (lts1 + lts2) − (gts1 + gts2) = 1 then
30:       return max (r1.hint, r2.hint)
31:    else
32:       return min (r1.hint, r2.hint)
33:    end if
34: end if
```

Fig. 3. Routine for aggregating report hints

of the order statistic must be greater than p. If both hints in the merged report are greater than p and the balance of moves up in the merged report is greater than one, it conservatively picks the maximum hint value (Line 22).

The inductive hypothesis ensures that this preserves the invariant because the interval between p and the maximum hint contains the intervals of both reports. If both hints in the merged report are greater than p and the balance of moves up in the merged report is equal to one, it picks the minimum hint value (Line 20). This also preserves the invariant because extreme moves cancel as with the omniscient observer. The case when there are more moves down is symmetric.

Another optimization reduces the number of iterations by sending all readings in the search interval to the base station when their number is known to be small. In the final iterations of the binary search, it is often more power efficient to request all sensor readings within the search interval than to continue

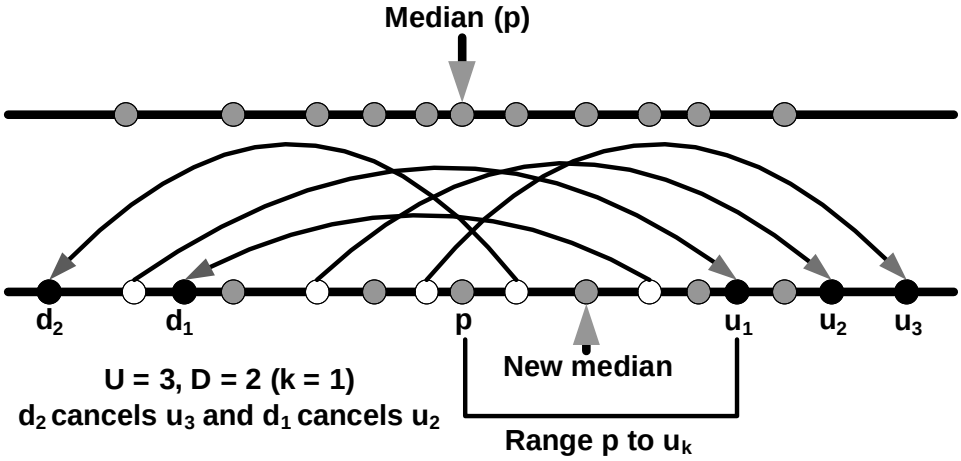


Fig. 2. Example showing how the values crossing the median can be canceled. The top line shows the distribution of sensor readings (grey circles) with the current median (p). The bottom line shows the new sensor values. If the new value differs from the old value, the old value is shown as a white circle and the new value is shown as a black circle.

searching. This only requires the base station to remember how many motes' readings are less than the lower bound of the search interval and how many are greater than the upper bound.

If the number of readings known to be within the search interval can fit in a single packet, the base station issues a request for all motes with readings within the search interval to forward their values up the tree. Because the number of values requested fit in a single packet, motes will send no more than one packet. In our prototype, 12 values fit in a single packet.

Once the values have all arrived at the base station, they are sorted. The base station then iterates through the sorted list, incrementing l until it reaches its maximum value. When this happens, the next value is the new value of the order statistic, p, and the number of values left in the list are added to g. Lastly, the base station broadcasts p to all motes, who store p and update their order state.

F. Generalizations

Currently, our POS prototype supports only persistent queries of a single order statistic at a time. However, our techniques can easily be generalized to compute more complete characterizations of the distribution of sensor readings. For example, it can be used to compute several order statistics simultaneously or to compute complete histograms.

For a single order statistic p, the base station maintains how many readings are less than, greater than, and equal to p. There are three intervals that any reading can fall into: $[0, p)$, $[p, p]$, and $(p, R]$, where R is the maximum value in the sensor's range. Reports simply summarize movement into and out of these intervals. We can use the same idea to compute several order statistics or to maintain a histogram.

For example to compute an histogram with n equal-length bins, the sensor range is split up into n intervals, $I_1, I_2, \ldots, I_n$, where $I_i = [\frac{R(i-1)}{n}, \frac{Ri}{n})$. Each mote's state is determined by the interval its reading falls into. Whereas POS reports contain four fields, persistent histogram reports would require two fields—$into_{I_i}$ and $outof_{I_i}$—for each interval I_i giving $2n$ total. These reports can be aggregated in the network like POS reports and transmissions can be suppressed as in POS. Leaf motes can suppress the transmission of their report if their new reading falls in the same interval as the last one and interior motes in the tree can suppress the transmission of reports when movement across intervals cancels out.

It is possible to compute several order statistics simultaneously in a similar way. One important difference between persistent histograms and persistent order statistics is that the interval boundaries are fixed by the choice of n for histograms and they vary for persistent order statistics. Therefore, the base station needs to maintain the interval boundaries for order statistics using an update protocol but there is no need for an update protocol when computing histograms.

POS can also be easily modified to answer top-k and bottom-k queries [8] that return the k readings with the largest or smallest values, respectively. For example to compute the top k readings, POS can compute a value p_k such that only k readings are greater than or equal to p_k using the techniques described above. Then it can issue a query for motes to return any readings greater than or equal to p_k. This approach provides an exact answer. Computing p_k has $O(1)$ communication cost per mote and collecting the top-k readings in the final query places a communication burden of $O(k)$ on the nodes close to the root of the tree, which is asymptotically optimal.

IV. Prototype Evaluation

We implemented a POS prototype and ran experiments to evaluate its performance. Our experiments compared the performance of the POS prototype, an aggregation service similar to TAG [4], and our implementation of Patt-Shamir's algorithm [7] (which had not been implemented before). We ran experiments to compare the communication overhead incurred by the three systems when computing a persistent query for the median sensor reading. To compare the overhead of computing order statistics and averages, we also measured

the overhead required to compute a persistent query for the average sensor reading using TAG [4]. The experiments ran on Harvard University's mote testbed (MoteLab). We also performed simulations using TOSSIM [9].

The experiments were designed to answer the following questions:

- How is the communication load distributed across motes when computing medians in the three systems?
- How does the communication load compare when computing averages and medians?
- How does POS behave across a spectrum of distributions and temporal evolution models of sensor readings?
- What effect do the POS optimizations have on communication load?

A. Systems studied

Before presenting experimental results, we provide an overview of the systems evaluated. All are implemented on TinyOS-1.x.

Average computes the average of the sensor readings periodically as in TAG [4]. It creates an ad-hoc spanning tree of the sensor network that is rooted at the base station. Every epoch motes aggregate their sensor readings up the tree and the base station collects the average.

SendReadings computes the median as in TAG [4]. Each epoch motes forward their sensor readings through the spanning tree to the base station. Each mote forwards its reading as well as those of its children. TinyOS packets carry a 29 byte payload, eight of which we used for an aggregation tree-related header. This left 21 bytes to aggregate as many readings as possible. Thus, for two byte readings each packet carried ten readings. If a mote received more than ten readings from its children, it sent the readings to its parent in multiple messages.

SendCount is our implementation of Patt-Shamir's [7] algorithm. It computes the median sensor reading periodically using a binary search. Every epoch the base station computes the network size, as well as the maximum and minimum sensor readings. These are computed as in TAG by performing aggregation up the spanning tree towards the base station. After this initial count, the base station performs binary search in the interval between the minimum and maximum readings. It computes the average of the minimum and maximum to generate an estimate for the median, which it broadcasts to the motes. Motes respond with reports that count the number of readings less than, equal to, and greater than this value. These reports are also aggregated up the tree towards the base station. The base station uses the aggregated report to determine the half of the interval containing the median and broadcasts the midpoint of the new interval. This process continues until the base station determines the median.

The *POS* prototype consists of under 4,000 lines of code residing in less than 50 nesC files. The POS implementation incorporates all optimisations discussed in this paper.

B. MoteLab Results

MoteLab is a wireless sensor network deployed in the Maxwell Dworkin building at Harvard University. It is composed of 26 Xbow MicaZ motes [11] with Atmel ATMEGA128L processors running at 7.3MHz, 128KB of read-only program memory, 4KB of RAM, and a Chipcon CC1000 radio operating at 433 MHz with an indoor range of approximately 100 meters. Each mote has an attached sensor board with photo, temperature, magnetometer, and various other sensors. For each trial, motes logged transmitted messages to their serial port, where the messages were stored in an SQL database. For the experiments, we used only the photo sensor.

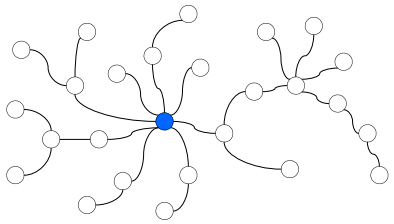


Fig. 4. MoteLab: Ad-hoc Tree and Grey Base Station

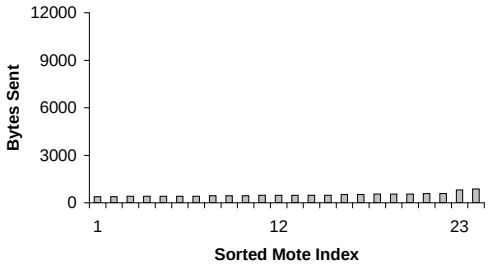
We ran POS, Average, SendReadings, and SendCount on the same 26 mote Network for an hour with an epoch of 1.5 minutes. Each system generated 40 samples of the median or average sensor reading during the experimental run. All systems computed exact samples of the median or average in these experiments.

The motes in all systems self-organized into a spanning tree rooted at the base station which was used to communicate sensor readings to the base station every epoch. Figure 4 shows an example spanning tree observed in one of the experimental runs. The base station is shaded. In this tree, the maximum branching factor was four and the depth was six.

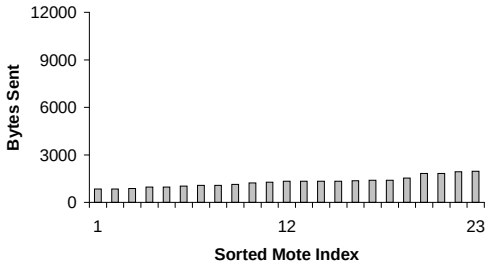
During each experiment, every mote recorded the total bytes sent, including broadcasts and retransmissions. We used the TinyOS Active Message (AM) layer [13] for communication. Each AM packet contains a five byte header, a 29 byte payload, and a two byte CRC footer. Therefore, packets have a *fixed length* of 36 bytes.

Figure 5 shows the sorted communication load of all motes under POS and Average. Since the mote photo sensor readings changed slowly and the observed range of values was narrow, POS' update protocol rarely ran and traffic suppression significantly reduced the number of bytes sent during the validation protocol. In Average, every mote transmitted reports to their parent every epoch. Since each packet is 36-bytes long, each mote was expected to transmit 1440 bytes over 40 epochs. Figure 5(b) shows that some nodes transmit more bytes. This is because of retransmissions. The TinyOS MAC layer for the MicaZ mote provides message acknowledgments, which we used to trigger retransmissions. The results show that POS induces a lower communication overhead than Average.

Figure 6 shows the sorted communication load of all motes for SendReadings and SendCount. These results show that the overhead induced by SendCount is significantly higher than the overhead induced by any other approach. At the beginning of each epoch, SendCount computes the max and min sensor

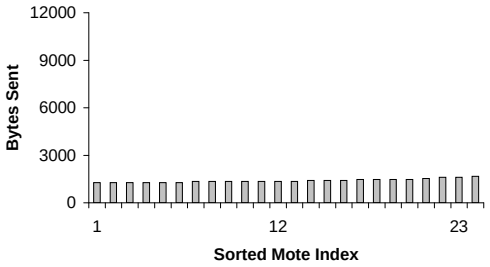


(a) POS

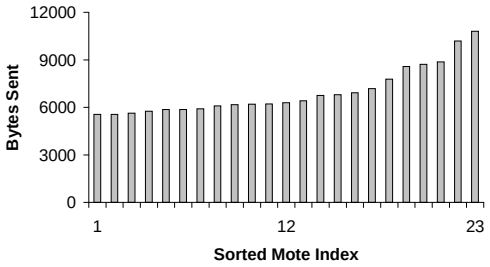


(b) Average

Fig. 5. MoteLab: POS and Average



(a) SendReadings



(b) SendCount

Fig. 6. MoteLab: SendReadings and SendCount

Approach	Max Msgs	Min Msgs
POS	33	18
Average	84	42
SendReadings	50	44
SendCount	423	272

Fig. 7. MoteLab: Message Transmission Statistics

readings, which requires each mote to send two packets. Then the base station broadcasts the estimated median and collects aggregate counts, which requires two more packets per mote. For some epochs, further rounds may be required. Hence, each mote is required to transmit at least four 36-byte packets during each epoch or 5760 bytes in 40 epochs. The additional messages are due to retransmissions.

The communication overhead induced by SendReadings is almost constant across all motes because packets are fixed size and no mote receives more than 10 readings in this small network (as shown in Figure 4). Therefore, each mote sends a single packet in each epoch or 1440 bytes over the 40 epochs. Variations are due to packet retransmissions. In this small network, SendReadings induces the same overhead to compute the median as Average induces to compute the average. However, SendReadings does not scale. Its performance would be significantly worse than Average for large networks.

These figures show that POS has the lowest communication overhead of all the approaches, followed by Average, SendReadings and SendCount. The maximum bytes transmitted by a mote under SendReadings or Average is twice the maximum bytes transmitted by a POS mote. The overhead of SendCount is even higher. The maximum bytes sent by a SendCount mote was nearly ten times the maximum bytes sent by a POS mote. POS is not only more efficient for the motes with the heaviest load. The Average, SendReadings, and SendCount motes with the least load still sent one and a half to 14 times more bytes than their POS counterparts.

In order to further understand the results, we also measured the maximum and minimum number of packets transmitted per mote in the different systems. These results are shown in Figure 7. The minimum number of messages sent by a mote in both Average and SendReadings is very close to the number of epochs as would be expected. The additional messages are due to retransmissions. The number of messages for SendCount is much higher because each mote must transmit at least four messages per epoch. In the absence of retransmissions, all motes would be expected to send the same number of packets. The POS results show the benefits of validation and transmission suppression. POS sends significantly fewer messages than all the other systems.

The photo readings we observed moved very slowly relative to the epoch length. After an initial period during which the sensors warmed up, the median reading did not change. Because of this, POS transmitted almost no messages in steady state. Each of the other approaches continued to send information back to the base station. One could lengthen the epoch period to reduce this overhead, but doing so would compromise accuracy. POS requires no such compromise. It provides accurate statistics at the lowest cost.

C. TOSSIM Results: Locality and Stability

The MoteLab results demonstrate that POS can offer significant communication savings relative to other approaches even in small sensor networks. Because SendReadings places an $O(N)$ load on motes near the base station, its overhead

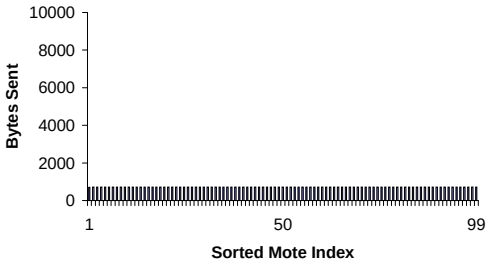
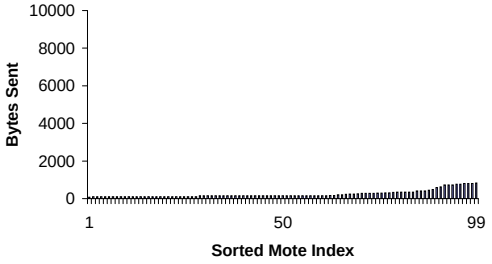
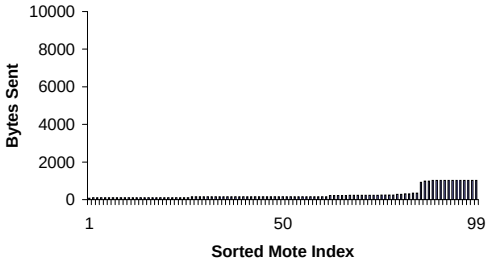


Fig. 8. TOSSIM: Average



(a) Binomial Distribution



(b) Uniform Distribution

Fig. 9. TOSSIM: POS with Stable Readings

should increase as the number of motes, N, increases.

To evaluate the systems on a larger network, we used the TinyOS simulator, TOSSIM [9]. TOSSIM simulations compile directly from TinyOS sourcecode. Each mote in the simulation had a radio with a radius of 30 meters. Links were asymmetric and messages received simultaneously were XORed to simulate interference. Unlike the motes used in the MoteLab experiments, the TOSSIM MAC layer does not provide message acknowledgments. Motes can still listen if others are transmitting before they send a packet. Since there are no retransmissions, some messages were lost but these losses did not affect the accuracy of the medians computed in the experiments. Our TOSSIM networks consisted of 100 motes arranged on a 300 meter by 300 meter grid. The simulations ran until the base station recorded 20 median or average values using an epoch of two minutes. As with the MoteLab experiments, the packet length was fixed at 36 bytes.

POS' communication overhead varies depending on both the distribution of the readings (*locality*) and the way they change over time (*stability*). In the MoteLab experiments, individual readings were highly correlated and changed little over the course of the experiment. Using TOSSIM, we evaluated POS

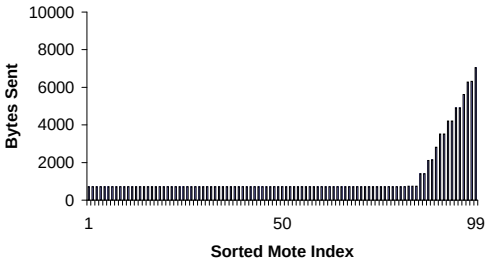
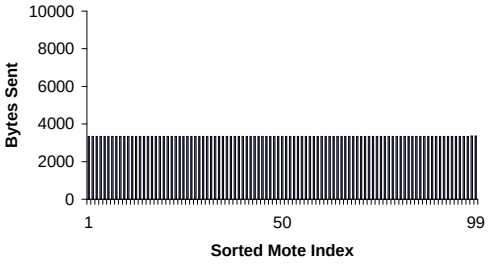
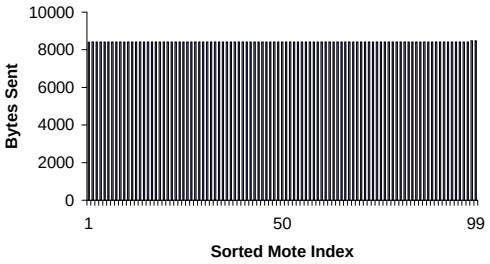


Fig. 10. TOSSIM: SendReadings

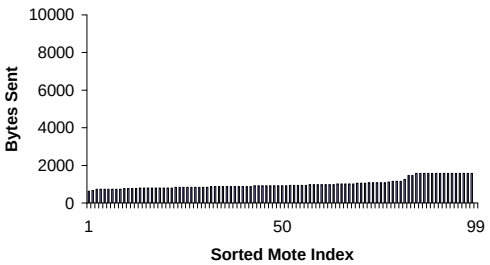


(a) Binomial Distribution

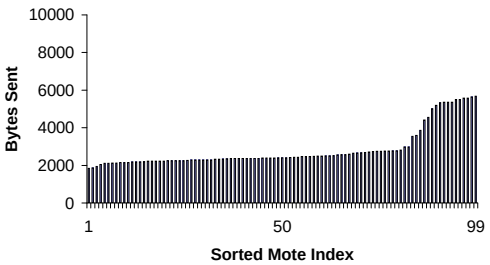


(b) Uniform Distribution

Fig. 11. TOSSIM: SendCount with Stable Readings

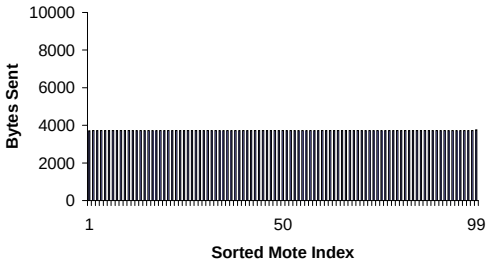


(a) Binomial Distribution

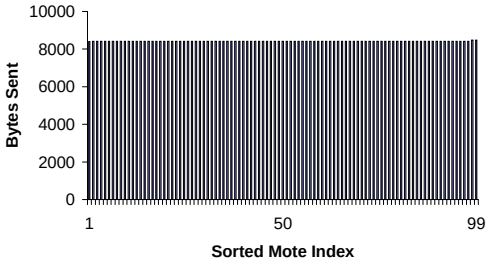


(b) Uniform Distribution

Fig. 12. TOSSIM: POS with Unstable Readings



(a) Binomial Distribution



(b) Uniform Distribution

Fig. 13. TOSSIM: SendCount with Unstable Readings

under more challenging conditions.

To control reading locality, motes chose their readings from one of two distributions. In one group of experiments, motes chose readings from a discrete, binomial distribution with success probability 0.5 and 100 trials. This provided locality around the value 50. In the other group of experiments, motes chose readings from a uniformly random distribution, which demonstrated no locality.

To control reading stability, we varied how readings changed from epoch to epoch. In one group of experiments, motes chose new readings after every epoch, independently of their previous reading. In other experiments, after an initial reading was chosen, motes chose subsequent readings by randomly perturbing the previous epoch's by -1, 0, or 1. We compared POS's communication overhead under each combination of stability and locality to Average, SendReadings, and SendCount.

Average and SendReadings perform the same regardless of locality or stability. Figures 8 and 10 show the distribution of bytes sent under Average and SendReadings, respectively. When running Average, each mote transmits a packet in every epoch. Each mote was expected to transmit 720 bytes in 20 epochs, as shown in Figure 8. When running SendReadings, most motes will also send just a single packet per epoch—720 bytes over the length of the experimental run. However, motes closer to the base station must send multiple packets. If a single mote is required to forward all 100 readings to the base station each epoch then it will send 10 packets per epoch, or 7200 bytes in total. Figure 10 demonstrates the higher load experienced by the motes closer to the base station.

Figure 9 shows the overhead of POS when readings are stable. POS performs very well under these conditions. When readings express locality (Figure 9 (a)), the most bytes sent by a POS mote is nearly ten times less than the most bytes sent by a SendReadings mote and is equal to the most bytes sent by an Average mote.

When readings do not express locality (Figure 9 (b)), the most bytes sent by a POS mote is nearly seven times less than the most bytes sent by a SendReadings mote. As in the MoteLab experiments, POS's benefit is not limited to the most burdened motes. The SendReadings motes who sent the fewest bytes had an overhead of nearly ten times the POS motes that sent the fewest bytes, regardless of distribution.

Unfortunately, without locality POS can be more expensive than Average. The most burdened POS node transmits over twice as many bytes as the most burdened Average node, but the least burdened motes in each approach transmit approximately the same number of bytes. We expect the readings of most deployments to exhibit some degree of locality.

Figure 11 shows the distribution of bytes sent under SendCount when readings are stable. With or without locality, SendCount is much more expensive than any of the other systems. The results without locality are worse because the interval of observed sensor readings is wider and, therefore, binary search requires more rounds to compute the median every epoch. The overhead introduced by SendCount when readings are not stable is shown in Figure 13. It is almost identical to the overhead observed with stable readings because SendCount does not implement validation or transmission suppression optimizations that exploit stability.

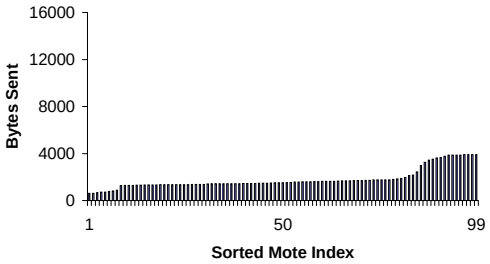
To evaluate POS in more demanding conditions, Figure 12 shows the communication overhead of POS when readings are not stable. These conditions are unlikely in real settings because it is unlikely for sensor readings to change independently each epoch.

POS performs well when readings exhibit locality (Figure 12 (a)) even when they are not stable. The maximum number of bytes sent by a POS node when readings follow the binomial distribution is four times less than the maximum number of bytes sent by a SendReadings node.

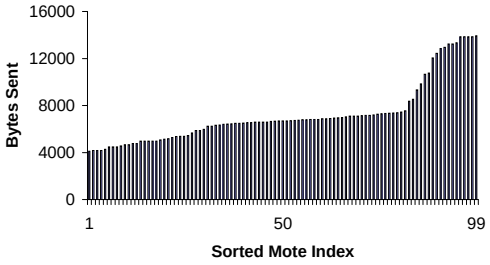
In the worst case, when readings exhibit neither locality nor stability, POS' advantage over SendReadings is less pronounced (Figure 12 (b)). However, POS is still 25% more communication efficient than SendReadings and the performance advantage would be larger for larger networks because POS scales better.

D. Optimizations

The final question we wanted to answer was how effective our more complex optimizations were at reducing message transmissions. Specifically, we were interested in in-network transmission suppression, sophisticated report hints, and opportunistically sending readings to reduce the number of rounds of binary search. To isolate the effect of each, we removed one optimization from POS at a time and reran our TOSSIM experiments. None of these optimizations significantly reduced communication when readings exhibited stability and locality because validation and local transmission



(a) POS



(b) POS + MaxHints

Fig. 14. TOSSIM: Stratified Perturbations

suppression are already so effective. This reduces the impact of the other optimizations in many common cases. However, the more complex optimizations did prove useful in more pathological environments.

For example, when readings were unstable and chosen from a uniform distribution, opportunistically sending readings back to the base station significantly shortened the runtime of the update protocol. When POS stopped sending readings back to the base station, the average number of rounds to recover the median with binary search rose from 3.76 to 5.59 (an increase of nearly 50%). It should be noted that the number of bytes sent was similar in both cases; most of these extra search rounds did not elicit responses from the motes because of our transmission suppression optimizations. However, reducing the number of rounds is important because it enables POS to reduce the minimum epoch users can specify in persistent queries.

In-network transmission suppression proved to be more useful when readings were unstable and chosen from a binomial distribution. In this case, the number of bytes sent increased without the optimization. Among motes with the greatest communication burden, in-network filtering removed approximately 18% of their load. As expected, nodes with the least burden (i.e. leaf nodes) did not benefit from in-network filtering at all, but they did benefit from our local transmission suppression optimization.

The sophisticated report hints did not provide benefit relative to the simpler strategy of sending the max and min movement hints (see SectionIII-E) in our previous experimental scenarios. Since we expect the more sophisticated hints to provide the most benefit when there is some geographical correlation in the changes of sensor readings, we applied a new distribution of sensor readings with stratified perturbations.

Readings were initially chosen from the same binomial distribution used in previous experiments. Each new epoch's reading was equal to the previous epoch's reading plus a perturbation. Motes' perturbations decreased exponentially the farther out they were from a disruptive center. We would like to have explored moving the disruptive center around the network, but this proved difficult.

The only approximation a mote has of its geographic coordinate is its level in the spanning tree. Thus, each mote's level in the spanning tree determined its degree of perturbation. The readings of motes closest to the base station varied more than those of far away motes. Specifically, perturbations were chosen uniformly from the range $[-(M-l)^2, (M-l)^2]$, where M was the maximum tree depth and l was a mote's tree depth. Thus, the base station's reading varied the most, while motes at depth M did not vary at all.

We ran POS under this distribution of sensor readings using our normal report hints and a more conservative approach of using the maximum and minimum of changed values as hints. Figure 14 shows the distribution of bytes sent by each approach to compute 20 median readings over a 100 mote network in TOSSIM. The maximum bytes sent by a mote using extreme hints was nearly two and one third times more than those sent using our report hints. This demonstrates that our report hints can effectively narrow the search scope by canceling extreme changes whenever possible.

V. Related Work

TAG [4] represents the most common model for collecting data from wireless sensor networks. Through persistent queries and in-network aggregation, TAG can power efficiently compute statistics like min, max, count, and average. Unfortunately, TAG requires motes to send all values back to the base station to compute the median reading.

One facet of TAG that is similar in spirit to POS, however, is its use of *hypothesis testing*. With hypothesis testing, motes attempt to infer their local reading's impact on the global state by snooping on the network. If the values a mote hears from its neighbors are similar to its own, it suppresses its transmission. Persistent order statistics give motes the precise global state rather than just a local approximation. This allows motes to make better decisions about the effect of their own reading.

In addition to TAG, several projects have recently looked at computing order statistics in sensor networks using approximations. One such system [5] uses compressing data structures called *q-digests*. Q-digests are tree-like structures that summarize how many values lie within pre-determined intervals of the reading range. Because q-digests are fixed-size and easy to aggregate, motes send $O(1)$ bytes per computation. Unfortunately, fixed-size q-digests must be relatively large to maintain acceptable accuracy. For example, a 50 byte q-digest gives between 20 and 25 percent error. The authors suggest reducing percent error to two percent by using 400 bytes messages. By comparison, POS also requires $O(1)$ bytes

per node transmitted, but needs less than the 29 byte packet payload provided by TinyOS to provide full accuracy.

A similar project applies techniques from data base research [6] to the problem of sensor network order statistics. This approach employs *quantile summaries* as well as multiple passes of the network to approximate order statistics. For a network of size n, this approach requires motes to send $O(log^2 n/\epsilon)$ values per computation, where $\epsilon < 1$ is the percent error. To compute the exact value, motes must send $O(log^3 n)$ values. In comparison, POS places an $O(1)$ transmission load on all motes and provides exact values.

Lastly, Patt-Shamir [7] describes a technique to compute order statistics using binary searches that is similar to our update protocol. This technique achieves $O(1)$ communication complexity per sensor node but our results show that the constants involved are large. It requires more messages to compute a median than sending all the sensor readings to the base station even for networks with as many as 100 nodes. A technique similar to Patt-Shamir's is presented in [14]. POS improves on these techniques by adding a validation protocol that can avoid the binary search in many epochs when computing persistent queries. Additionally, it uses knowledge of the previous sample of an order statistic or the value computed during the previous iteration of a binary search to suppress communication and reduce the number of iterations of binary search. Our results show that these improvements have a large impact on performance.

VI. Conclusion

Wireless sensor networks are used to monitor a wide range of environments. Since energy efficiency is critical in these deployments, almost all use a combination of persistent queries and in-network aggregation to collect data and conserve energy. These techniques work well for statistics such as max, min, and average, but not for order statistics such as median. Current techniques for computing order statistics either forward all sensor readings to a collection point or calculate approximate results. Neither provides both energy efficiency and accuracy. This is unfortunate because order statistics are more resilient to faulty sensor readings than max, min, or average.

To ease this tension between resilience and energy efficiency, we propose POS, an energy-efficient, in-network service that computes accurate order statistics. POS returns a stream of periodic samples of any order statistic. It initially computes the value of the order statistic and then periodically runs a validation protocol to determine whether the value is still valid. If not, it uses an optimized binary search to determine the new value and then resumes periodic validation. POS uses in-network aggregation and transmission suppression to achieve $O(1)$ communication complexity per node.

We presented the design and implementation of POS and an evaluation of the POS prototype using Harvard University's MoteLab and the TOSSIM simulator. Results from these experiments demonstrate that POS can compute order statistics accurately while consuming less energy than the best techniques to compute averages in common cases.

References

[1] D. Culler, J. Demmel, G. Fenves, S. Kim, T. Oberheim, and S. Pakzad, "Structural health monitoring of the Golden Gate Bridge," http://www.eecs.berkeley.edu/~binetude/ggb/.

[2] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application driver for wireless communications technology," in *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001.

[3] G. Werner-Allen, M. Welsh, J. Johnson, M. Ruiz, and J. Lees, "Monitoring volcanic eruptions with a wireless sensor network," http://www.eecs.harvard.edu/~werner/projects/volcano/.

[4] S. Madden, M. Franklin, and J. Hellerstein, "TAG: A Tiny AGgregation service for ad-hoc sensor networks," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[5] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *Proceedings of the 2nd Conference on Embedded Networked Sensor Systems*, Baltimore, MD, November 2004.

[6] M. Greenwald and S. Khanna, "Power-conserving computation of order-statistics over sensor networks," in *Proceedings of the 23rd Symposium on Database Principles*, Paris, France, June 2004.

[7] B. Patt-Shamir, "A note on efficient aggregate queries in sensor networks," in *Proceedings of the 23rd Symposium on the Principles of Distributed Computing*, St. John's, Canada, July 2004.

[8] A. Silberstein, R. Braynard, C. Ellis, K. Munagala, and J. Yang, "A sampling-based approach to optimizing top-k queries in sensor networks," Duke University, Tech. Rep., March 2005, cS-2005-01.

[9] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications," in *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks, SPOTS track*, Los Angeles, CA, November 2003.

[10] G. Werner-Allen, P. Swieskowski, and M. Welsh, "MoteLab: A wireless sensor network testbed," in *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks, SPOTS track*, Los Angeles, CA, April 2005.

[11] "Crossbow technology inc." http://www.xbow.com/.

[12] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proceedings of the First Symposium on Networked Systems Design and Implementation*, San Francisco, CA, March 2004.

[13] P. Buonadonna, J. Hill, and D. Culler, "Active message communication for tiny networked sensors," http://www.tinyos.net/papers/ammote.pdf.

[14] A. Negro, N. Santoro, and J. Urrutia, "Efficient Distributed Selection with Bounded Messages," *IEEE Transactions on Parallel Distributed Systems*, vol. 8, no. 4, Apr. 1997.