View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Harvard University - DASH

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Energy and Storage Reduction in Data Intensive Wireless Sensor Network Applications

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation	Chang, Stephen, Adam Kirsch, Michael Lyons. 2007. Energy and Storage Reduction in Data Intensive Wireless Sensor Network Applications. Harvard Computer Science Group Technical Report TR-15-07.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:25620450
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Energy and Storage Reduction in Data Intensive Wireless Sensor Network Applications

Stephen Chang, Adam Kirsch, Michael Lyons

TR-15-07

Computer Science Group
Harvard University
Cambridge, Massachusetts

Energy and Storage Reduction in Data Intensive Wireless Sensor Network Applications

Stephen Chang Adam Kirsch* Michael Lyons†

Harvard School of Engineering and Applied Sciences
Cambridge, MA 02138
{stchang,kirsch,mjlyons}@fas.harvard.edu

Abstract

Recording large data sets in current wireless sensor networks is difficult if not impossible, as sensor network nodes feature extremely limited capabilities. Meager amounts of memory limit storage and slow network transmission rates limit data transfer. Even if enough data is recorded, network lifetime may be brief due to energy consumption. In this paper, we discuss using Bloom filters to reduce the memory and network transmission requirements for sending or storing large amounts of data. Using our approach, we can gather more information from our wireless sensor network and simultaneously extend the network lifetime.

1 Introduction

Recording large amounts of information in wireless sensor networks is a challenge. Sensor networks are designed to monitor an area, sensing temperature, objects, or other aspects of the environment for long periods of time. The nodes in these networks, called *motes*, each contain a processor, memory, a wireless network radio, and a battery. Although motes are full computers, they are constrained in many ways. Network transmissions are typically slow and power-hungry, memory is small, the processor is sluggish, and the battery supplies a limited amount of energy. Data intensive applications that send results to a central server are particularly constrained due to the large amount of storage space required to buffer readings, as well as the time and energy needed for wireless transmission. Alternatively, applications that store readings locally on the motes may not need to continuously transmit data, but since motes must store readings for a prolonged period of time, memory is still a precious resource.

We introduce a framework for addressing the limitations of both of these data intensive application classes. Our system is based on Bloom filters, which are simple randomized data structures for set membership. For those not familiar with Bloom filters, we review them in Section 2. When appropriate, we also compress our Bloom filters using Golomb-Rice coding (e.g. [20, 26, 29]). Using our approach, we are able to reduce the amount of memory used to store a set of large elements

*Supported in part by an NSF Graduate Research Fellowship, NSF grant CNS-0721491, and grants from Cisco Systems and Yahoo! Research.

†Supported in part by NSF grant CNS-0330244

by up to 68%, at the cost of a small, manageable false positive rate. This reduction in storage naturally leads to a 68% decrease in transmission-related energy. This maximum savings applies to those motes that send the largest amount of data. Thus, our system significantly extends the lifetime of the motes that would otherwise deplete their batteries the fastest, resulting in an even further increase in network lifetime. Additionally, when we compress our Bloom filters, we still obtain excellent results over a wide range of settings. Furthermore, the mote processing time required to implement our system is small enough to be sufficient for a wide range of applications; we give some potential examples in Section 3.

In essence, our system provides a summary of sensor network readings in the form of a Bloom filter. In contrast, many projects use *aggregation* to create different sorts of summaries of this data. Aggregation attempts to interpret and summarize data inside the sensor network to reduce the amount of data transmitted to a central server. For example, consider a query to calculate the average temperature sensed by all motes in the network. Determining the average inside the network reduces transmissions to the server by only reporting the average temperature instead of all readings. This approach has demonstrated energy reductions in several applications (e.g. [9, 12, 17, 18]). However, there are many useful potential calculations that either require knowledge of the entire set of readings or computational abilities beyond mote processors. In these cases, aggregation is not appropriate, and our approach is much more feasible.

We note that some sensor network projects currently use Bloom filters or similar data structures. For instance, the above averaging example may return an incorrect result if the same temperature readings are used multiple times. This situation may occur as a result of network retransmissions or routing problems. Considine et al. [5] and Nath et al. [23] use FM sketches to track which readings have been included in the averaging aggregate to prevent reusing them. Ghose et al. [10] use Bloom filters to locate data stored inside the sensor network. Hebden and Pierce [13] use Bloom filters for routing to optimize network traffic. Each approach uses Bloom filters to assist in data management but not to store readings directly. In contrast, we propose using Bloom filters to store readings. Our implementation is also insensitive to duplicate readings and can be used to implement the designs cited above.

Prior work has also investigated compression to reduce storage and transmission demands in wireless sensor networks. For instance, Ganesan et al. [9] and Cianco et al. [4] describe the use of wavelet compression in sensor networks. Kimura and Latifi [16] summarize other compression algorithms, such as a method similar to MPEG-2. We implement Golomb-Rice coding in this paper due to its low computational requirements and its effectiveness for compressing Bloom filters.

2 Bloom Filter Review

We begin by reviewing the fundamentals of Bloom filters, based on the presentation of the survey [1], to which we refer for further details. A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from a large universe U consists of an array of m bits, initially all set to 0. The filter uses k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$, where it assumed that these hash functions map each element in the universe to a random number uniformly over the range. While the randomness of the hash functions is clearly an optimistic assumption, it appears to be suitable in practice [8, 24]. (Indeed, we shall see another example of this in our assessments in Sections 5 and 6.) For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. (A location can be set to 1 multiple times.) To check if an item y is in S, we check whether all bits $h_i(y)$ are set to 1. If not,

then clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S. Of course, it is possible that all of these bits are set to 1 even if $y \notin S$, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set, or the *false positive probability*, can be estimated in a straightforward fashion, given our assumption that hash functions are perfectly random. After all the elements of S are hashed into the Bloom filter, the probability that a specific bit is still 0 is

$$p' = (1 - 1/m)^{kn} \approx e^{-kn/m}.$$

In this section, we generally use the approximation $p = e^{-kn/m}$ in place of p' for convenience.

If ρ is the proportion of 0 bits after all the n elements are inserted in the table, then conditioned on ρ the probability of a false positive is

$$(1 - \rho)^k \approx (1 - p')^k \approx (1 - p)^k = \left(1 - e^{-kn/m}\right)^k.$$

These approximations follow since $\mathbf{E}[\rho] = p'$, and ρ can be shown to be highly concentrated around p' using standard techniques. It is easy to show that the expression $\left(1 - e^{-kn/m}\right)^k$ is minimized when $k = \ln 2 \cdot (m/n)$, giving a false positive probability f of

$$f = \left(1 - e^{-kn/m}\right)^k = (1/2)^k \approx (0.6185)^{m/n}.$$

In practice, k must be an integer, in which case one of the neighboring values is optimal.

This analysis provides us (roughly) with the probability that a single item $z \notin S$ gives a false positive. In fact, this probability acts like a *rate*. That is, if we choose a large number of *distinct* elements not in S, the fraction of them that yield false positives is approximately f with extremely high probability. This result follows immediately from the fact that ρ is highly concentrated around p', and for this reason, the false positive probability is often called the *false positive rate*.

It is very important to note that a Bloom filter gives a representation of a set with a storage cost in terms of bits per item, regardless of the actual size of the items. Thus, if items are large, there is an almost immediate storage savings in using a Bloom filter to represent a set instead of actually storing the items themselves. We make frequent reference to this observation throughout this work.

Finally, Bloom filters have another property that is essential for our system. Suppose that we have two Bloom filters of the same size with the same hash functions, representing two different sets S_1 and S_2. Then it is easy to see that the corresponding Bloom filter for the set $S_1 \cup S_2$ can be obtained by taking the bitwise OR of the filters corresponding to S_1 and S_2. By induction, the Bloom filter corresponding to a union of sets is the same as the bitwise OR of the Bloom filters corresponding to each of those sets. Thus, to construct a Bloom filter for a union of sets, it suffices to merely have a Bloom filter for each set, as opposed to knowing each of the sets with certainty. This observation is the primary reason that we use Bloom filters in this work, instead of some other data structure for set membership. Indeed, the system that we introduce in this paper is built almost entirely on this fact.

3 Reference Applications

We now briefly discuss a few general applications that could use our approach for improved scalability, while using error management techniques to simultaneously provide accurate results and

efficiently minimize memory use and transmissions. Any of these applications can either store readings locally or continuously send readings to a central server. Distributed local storage is ideal for applications that only require occasional query requests sent over the network. Systems requiring many queries or faster response times should continuously send readings to a server for centralized storage.

Before continuing, recall that a Bloom filter query only reveals whether one particular element is in the set represented by the filter, with the potential for a false positive. Since false positives accumulate over multiple queries, it is important to develop techniques for managing the overall error. There are at least two independent methods for keeping this aggregate error manageable. First, we can simply reduce the number of queried elements. By keeping this *query window* small, we also keep the total error small. Second, post-query validation on the server can remove suspicious readings that are likely to be false positives. Thus, by managing the query window and validating query results, we can, in principle, keep false positives satisfactorily low.

3.1 Object Tracking

Wireless sensor networks have been previously used to track the location of objects (e.g. [27]). Object tracking is useful in many settings, such as monitoring merchandise in a warehouse with RFID tags [30]. Bloom filters provide an excellent storage method for this class of applications. For example, assuming that each mote in the network sees 5 objects on average, a back-of-the-envelope calculation shows that an 8KB Bloom filter could allow the network to scale up to 1300 motes. If an application simply wants to know whether an object was detected, the tag ID is used as the element and stored in the Bloom filter. If the location of the object is required, the mote's unique identifier can be combined with the tag ID and inserted into the filter. The server can determine the object's location because it knows which mote observed the object. However, the second approach may result in more error because a full sweep of each mote and ID combination would be required to detect all tags, enlarging the query window.

Several methods can be used to reduce this error. First, special motes at entrances and exits could report tag IDs entering or leaving the network area without Bloom filters. With this knowledge of which IDs are present in the network, the query window can be reduced from the universe of all IDs to the IDs in the network area. Second, multiple motes will typically see a tag over a period of time. A Bloom filter claiming to find an RFID tag spontaneously in the middle of a warehouse is likely erroneous.

3.2 Mote Status

Bloom filters provide large-scale wireless sensor networks an efficient framework for reporting network-wide status. For instance, suppose that the sensor network operator wants to monitor the battery levels of each mote in the network. Each mote can periodically add its unique identifier to the Bloom filter if its battery is low. The final Bloom filter delivered to the operator contains the set of all motes with low batteries. A back-of-the-envelope calculation shows that if we use an 8KB Bloom filter and assume that 10% of motes at any given time require battery replacement, then the network can scale up to 65,000 motes. The query window can be easily reduced since the operator knows which unique identifiers are present in the network and can limit queries to just those motes. Post-query error reduction can be performed by removing motes that sporadically report low batteries.

This model can be augmented in various ways. Support for multiple levels of low battery could be added to prioritize battery replacement. Motes could query the Bloom filter for neighboring motes when relaying Bloom filters to determine if any have low batteries. If so, they could instruct neighboring motes to avoid routing data through the mote with a low battery to avoid total energy exhaustion and prolong network longevity. The model could also be extended to increase scalability for current status-based work in heart rate monitoring for a large population [19], volcano tremor severity over an area [28], or the structural integrity of a bridge [15].

3.3 Localization

Bloom filters could compliment prior work for the discovery of a sensor network's physical layout (e.g. [11,25,27]). For instance, a mote could insert elements by combining its unique identifier with the unique identifier of each mote nearby. When the Bloom filter containing these unique identifier pairs reaches the central server, a loose topological map of the network could be built. Assuming each mote is within communication range of two other motes on average, a back-of-the-envelope calculation shows that an 8KB Bloom filter would allow the system to scale to 3250 motes. This map could be refined by including a wireless link quality level between element pairs to estimate the distance between motes. Like the mote status application, the query window could be reduced to just the motes known to exist in the network. Post-query error reduction could verify reciprocal pairings, as a pair is likely to be valid only if reported by both motes.

4 System Design

Our system supports TinyOS 1.1.15 [14] and is designed for the Moteiv Tmote Sky hardware platform. Our implementation supports Bloom filters with bit arrays of up to 8KB. We also require an additional 351 bytes of RAM for variables and 7016 bytes of instruction code stored in ROM. Larger Bloom filters could be supported for platforms with more memory, and we give some proof of concept results to that effect in Section 5. We support 32-bit items, although the implementation could be modified to support larger elements. Indeed, the use of larger elements would likely only improve the relative performance of our system over an approach that explicitly stores items.

4.1 System Overview

Our system is designed to support either of two models of Bloom filter storage: centralized and distributed.

4.1.1 Centralized Storage

The centralized storage model uses Bloom filters to efficiently transmit readings to a central server. In this model, a mote network arranged in a tree publishes readings through the network in Bloom filters. First, a leaf mote inserts each reading into a new Bloom filter. This Bloom filter is compressed and transmitted to the parent mote. The parent mote uncompresses and merges all Bloom filters received from its children. It then inserts its own readings into the Bloom filter, recompresses the filter, and sends it to the grandparent mote. In this way, the Bloom filters merge all the way to the root where they are transmitted to a central server as one master Bloom filter. The server

then queries this master filter to determine readings and perform post-query error reduction if applicable. Here, each mote's local Bloom filter is erased following transmission to ensure readings are not retransmitted. Note that only the server is able to make queries into past readings because mote Bloom filters are regularly erased.

4.1.2 Distributed Storage

The distributed model uses Bloom filters to efficiently store readings locally on the motes. A central server performs queries by flooding the sensor network with a query request. Each mote performs a query on its local Bloom filter and responds with results. A Bloom filter can be used to send the response if it is sufficiently large. Aggregation techniques could also be used to reduce the response transmission size.

4.2 Bloom Filter Hash Functions

While Bloom filters are fairly straightforward to implement, the choice of the hash functions to use is a significant design decision. We use the following hash family of Dietzfelbinger et al. [7], who show that it is universal in the sense of Carter and Wegman [2]. For $\ell \in \{0, \ldots, 31\}$, we define $\mathcal{H}_\ell = \big\{h_{a,\ell} : 0 < a < 2^{32} \text{ and } a \text{ is odd}\big\}$, where $h_{a,\ell} : \big\{0, \ldots, 2^{32} - 1\big\} \to \big\{0, \ldots, 2^\ell - 1\big\}$ is defined by

$$h_{a,\ell}(x) = \lfloor (ax \bmod 2^{32})/2^{32-\ell} \rfloor \quad \text{for} \quad 0 \le x < 2^{32}.$$

Every Bloom filter that we consider has size $m = 2^\ell$ (bits) for some $\ell \in \{0, \ldots, 31\}$, and so we choose our hash functions independently and uniformly at random from $\mathcal{H}_\ell$. (In our implementation, we choose the values for a in advance and hard code them into the system.)

There are two main reasons to use this hash family. First, each $h_{a,\ell}(x)$ can be easily evaluated with a single 32-bit multiplication and a shift, so the hash functions are always very fast. This speed is critical for us, since motes have extremely limited computational power. Second, the universality property shown in [7] suggests that, at least in practice, this hash family yields probabilistic behavior similar to that obtained by choosing a fully random hash function. Since all of our theoretical analysis is based on the assumption that the hash functions are fully random, we need the latter property to ensure that those results are relevant to the actual performance of the system.

4.3 Bloom Filter Transmission

We now turn towards the issue of transmitting a Bloom filter. Since the energy and computational resources of the motes are extremely limited, we need a method that:

- compresses a Bloom filter before transmitting it, to minimize the amount of data that the motes must send,
- ensures that this compression and the corresponding decompression require little computational overhead, so that all motes can perform the operations required of them in the time allotted, and
- allows compression and decompression to occur on-the-fly during transmission to minimize memory overhead, so that we may use almost all of a mote's memory for a single Bloom filter (to minimize false positives).

In particular, since motes have limited memory, it is not practical for us to proceed as in [21], considering tradeoffs between the uncompressed and compressed sizes of the filter.

To see how we should compress a Bloom filter, we make the heuristic assumption that all of its bits are independent. Adopting the notation of Section 2, each bit of the filter is 1 with probability $1 - p'$, which is at most $1/2$ if the filter is properly configured. In this case, we can think of the filter as containing many *runs*, where each run consists of zero or more 0's, followed by a 1. At least intuitively, the joint distribution of the lengths of these runs should be similar to the joint distribution of independent $\mathrm{Geom}(1 - p')$ random variables. (Here $\mathrm{Geom}(q)$ is the geometric distribution with parameter q, defined by $\mathbf{Pr}(\mathrm{Geom}(q) = i) = q(1 - q)^i$ for $i = 0, 1, \ldots$.)

This line of thinking suggests that we should try an encoding scheme designed to handle sequences of independent and identically distributed $\mathrm{Geom}(1-p')$ random variables. We use Golomb-Rice coding (e.g. [20, 26, 29]), which is a very efficient prefix coding scheme designed specifically for this setting. The scheme works by fixing some $M \geq 1$ that is a power of 2, and encoding a value $x \geq 0$ by a sequence of $\lfloor x/M \rfloor$ zeros followed by a one, concatenated with the $\log_2 M$-bit binary representation of $x \bmod M$. This gives a computationally simple prefix code; to encode a sequence of values, we can just concatenate the encodings of the values, and the decoding algorithm is straightforward and extremely fast. Furthermore, no actual multiplication or division is required because M is a power of 2; all of those operations can be implemented through bit shifts.

We choose M to minimize the expected length $L_{M,p'}$ of the encoding of a $\mathrm{Geom}(1-p')$ random variable. We do this by writing an expression for $L_{M,p'}$ and numerically determining the values of p' for which $L_{2^i,p'} = L_{2^{i+1},p'}$, for $i = 0, \ldots, 7$ (after $i = 7$, the threshold values of p' are so close to 1 that having threshold values for larger i does not noticeably improve the performance for our application); see, for example [20]. We pre-compute these threshold values and hard code them into our application.

Now, to send a Bloom filter, we estimate $1 - p'$ by computing the fraction of bits in the filter set to 1, and then we pick the corresponding pre-computed value of M. (Actually, since the size m of the filter is hard-coded into the implementation, we can represent the threshold $1 - p'$ values for M as numbers of 1's in the filter, which allows us to determine M without any floating point arithmetic.) We think of the Bloom filter as a sequence of runs of zeros, and transmit the corresponding Golomb-Rice encoding of the lengths of those runs, along with the value of M. Note that if we use this scheme with $M = 1$, then we just send the original Bloom filter.

As we shall see in Section 5.2, this gives very effective compression, particularly when the number of ones in the Bloom filter is fairly small, as is the typical case in our application. (As an aside, the reason that we encode run lengths of zeros instead of run lengths of ones is because a properly configured Bloom filter almost certainly has more zeros than ones, so it is always preferable to encode run lengths of zeros; this can be seen through a direct analysis of the expected length of an encoded symbol, as in, for example, [20].) Also, this method is clearly computationally efficient. Furthermore, it is easy to see that we can implement it so that we can decompress and merge several transmissions on-the-fly, while simultaneously adding new items to the filter. (We evaluate this claim precisely in Section 6.2.) Thus, the scheme satisfies all three of our desiderata.

5 Evaluation: Off-Mote Simulations

In this section, we evaluate the performance of our system on a standard PC, using the standard Java pseudorandom number generator whenever random values are needed. For illustrative pur-

Number	m	n	k	f
1	65536 (8KB)	6500	7	$7.87 \times 10^{-3} < 1\%$
2	65536 (8KB)	4500	10	$9.15 \times 10^{-4} < 0.1\%$
3	65536 (8KB)	3000	15	$2.77 \times 10^{-5} < 0.01\%$
4	131072 (16KB)	13500	7	$9.44 \times 10^{-3} < 1\%$
5	131072 (16KB)	9000	10	$9.15 \times 10^{-4} < 0.1\%$
6	131072 (16KB)	6500	14	$6.20 \times 10^{-5} < 0.01\%$

Table 1: Bloom filter configurations

poses, we consider several Bloom filter configurations in our error evaluations. In particular, we consider Bloom filter sizes of 8KB and 16KB. We are able to implement the 8KB constructions on current mote hardware, and expect the 16KB constructions to become practical in the near future. For each of these two Bloom filter sizes, we consider three configurations, designed to have false positive probabilities just under 1%, 0.1%, and 0.01%. These configurations are determined from the analysis in Section 2, and are given in Table 1, using the notation from that section.

5.1 Bloom Filter False Positives

We begin by testing our Bloom filter configurations to ensure that our implementation behaves as predicted by the theoretical analysis in Section 2. In particular, we show that even using the simple hash functions described in Section 4.2, the false positive probabilities for our Bloom filters are close to the theoretical values. Furthermore, we show that these false positive probabilities act like rates, in the same sense as in Section 2: if we initialize one of our Bloom filter configurations to represent a set S of the appropriate size and then query the filter on many elements not in S, the fraction of false positives is very likely to be very close to the false positive probability.

We start by estimating the false positive probabilities for each of the Bloom filter configurations in Table 1. For a particular configuration, we let n denote the number of items supported and f denote the theoretical false positive probability. We generate an estimate $\hat{f}$ of f by instantiating the Bloom filter 1,000 times, each time populating it with a randomly chosen set $S \subseteq \{0, \ldots, 2^{32} - 1\}$ of n elements and then querying it for $\lceil 10/f \rceil$ randomly chosen items not in S, and setting $\hat{f}$ to be the observed fraction of false positives. In addition, in an effort to make sure that the random nature of S and the queries does not corrupt our results, we also experiment with some deterministic choices. In particular, we consider choosing some $i \in \{1, \ldots, 16\}$, letting $v_i = (0, i, 2i, \ldots, (n + \lceil 10/f \rceil - 1)i)$ denote the vector with *stride* i, and always choosing S to be the first n elements of v_i and the queries to be the remaining elements.

The results are shown in Table 2, where $\hat{f}$ denotes the estimate obtained by choosing S and the queries randomly, and $\hat{f}'$ denotes the result obtained from using stride 8. The results for the other strides considered are similar. Clearly, the estimates show that our Bloom filter implementation has a false positive probability close to the theoretical value f.

Next, we show that the false positive probability acts like a rate for our Bloom filter implementation, as it does for the theoretical construction where hash functions are fully random. As in Section 2, this fact is significant because it tells us that, in practice, different elements not in

Number	Items	f	$\hat{f}$	$\hat{f}'$
1	6500	7.87×10^{-3}	7.86×10^{-3}	6.34×10^{-3}
2	4500	9.15×10^{-4}	9.22×10^{-4}	8.73×10^{-4}
3	3000	2.77×10^{-5}	2.73×10^{-5}	3.04×10^{-5}
4	13500	9.44×10^{-3}	9.39×10^{-3}	7.44×10^{-3}
5	9000	9.15×10^{-4}	9.15×10^{-4}	8.02×10^{-4}
6	6500	6.20×10^{-5}	6.26×10^{-5}	6.72×10^{-5}

Table 2: Observed False Positive Probabilities

the set represented by the filter yield false positives more or less independently. Indeed, if the false positive probability did not act like a rate, then there might be a reasonable chance for us to be so unlucky in our choices of hash functions that a large fraction of the universe yields false positives for the filter. We show that this is not the case, and that, as before, our Bloom filter implementation behaves similarly to the theoretical prediction.

We use the following test to determine whether the false positive probability acts like a rate for a particular Bloom filter configuration. We conduct 10,000 independent trials, where each trial consists of instantiating the Bloom filter, populating it with a set S with n items, and querying it for $\lceil 10^3/f \rceil$ items not in S (where S and the queries are generated in the same way as in previous experiments concerning the false positive probability). For each trial, we observe the fraction of queries that yield false positives, and we look at the distribution of this fraction across all 10,000 trials. If this distribution is concentrated around its mean, then we say that the false positive probability acts like a rate.

Since these experiments are very time consuming (especially when f is small), we only consider the Bloom filter configurations in Table 2 that are designed to give false positive probability just under 1% (numbers 1 and 4). The results for configuration 1 when S and the queries are generated randomly (as in the previous batch of experiments) are shown in Figure 1. The results for configuration 4 and strides $1, \dots, 16$ are similar. We conclude that the false positive probability acts like a rate in our implementation.

5.2 Bloom Filter Compression

We now turn our attention to the Bloom filter compression scheme discussed in Section 4.3. We evaluate the effectiveness of the compression through simulation, deferring the analysis of an actual implementation until Section 6, where we take a closer look at some additional lower level issues.

Consider some fixed Bloom filter configuration. Adopting the notation of Section 2, we model each bit of the filter as being 0 independently with probability $p = e^{-kn/m}$. (As noted in Section 4.3, this assumption is heuristic even if we assume that the hash functions are fully random, but it suffices for our purposes.) In this case, Shannon's source coding theorem (e.g. [6]) implies that the smallest achievable expected compressed size of the Bloom filter is about $mH(p)$ bits, where $H(p) = -p \log_2 p - (1-p)\log_2(1-p)$ is the binary entropy function. For this reason, we use the (Shannon) entropy $mH(p)$ of the filter as our benchmark for assessing the effectiveness of our compression scheme.

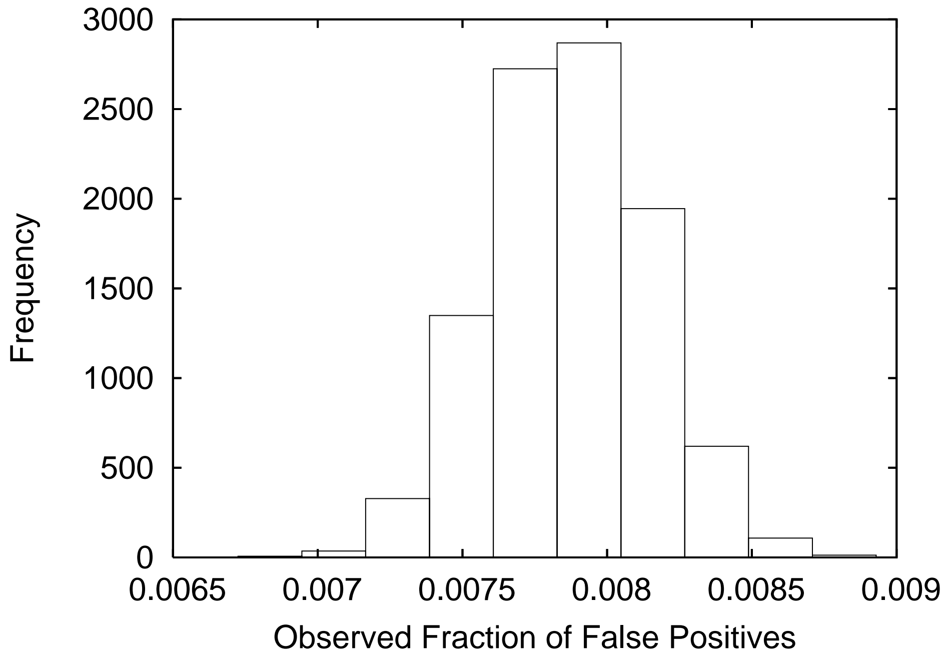

Figure 1: Histogram of observed fractions of false positives probabilities for Bloom filter configuration 1.

We are now ready to evaluate our compression scheme through simulation. For a particular Bloom filter configuration, we repeat the following 1,000 times. We instantiate the Bloom filter, populate it with a set S smaller than the maximum size supported by the filter, and measure the compressed size of the filter. We generate the set S using both the random method and the stride method discussed in Section 5.1.

In Figure 2, we compare the average compressed size for Bloom filter configuration 2 from Table 1 to its corresponding uncompressed size and estimated entropy, where S is generated according to the random method. The corresponding figures for the other Bloom filter configurations and the stride method are similar. The standard deviations of the compressed sizes are very small (about 0.1% of the mean), and so we omit error bars from the figure. (The standard deviations are slightly larger for the stride method, but still on the order of 1% of the mean.) Thus, our compression scheme is very effective, especially when the filter is sparsely populated, which is the most common case for Bloom filter transmission in the centralized storage model discussed in Section 4.1.1; we discuss this in detail in Section 6.1.

6 Evaluation: On-Mote Simulations

In this section, we report the results of simulations of our system implementation. (Recall that our implementation supports TinyOS 1.1.15 and runs on the Moteiv Tmote Sky hardware platform.) Here, the set of inserted elements is always determined using the random method of Section 5.

In our simulations, the motes are instructed via PC serial link to insert elements, compress, and merge compressed Bloom filters. The results of these Bloom filter operations are returned via

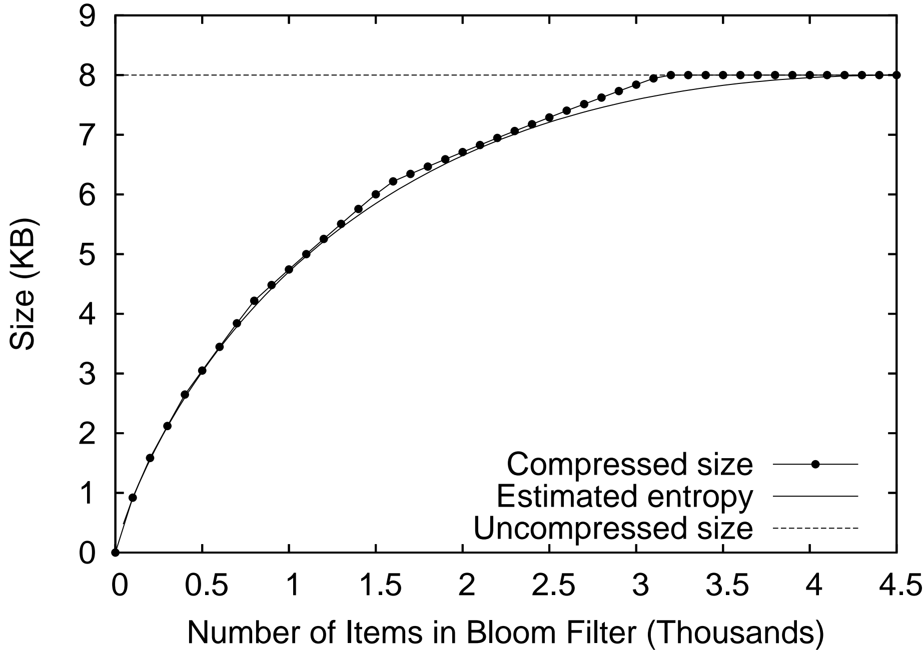

Figure 2: Effectiveness of compression for Bloom filter configuration 2.

the serial link and validated. The elapsed time for each of these operations, accurate to the nearest microsecond, is recorded on the mote, and also reported via serial link. We estimate mote processor energy consumption based on elapsed time and power figures from the Tmote Sky data sheet [22].

We assume an average data transmission rate of 40kbps, based on the results of [3]. Note that this data rate only includes application-layer data, not transmission overhead such as error detection or routing information. We also use radio power figures from the Tmote Sky data sheet to estimate energy requirements for network transmissions. Combining our estimates for computational and transmission energy use gives us the total system energy estimates given below.

6.1 Transmissions

We start by comparing the transmission sizes for the 8KB Bloom filter configurations in Table 1 against transmission size resulting from a naive sequential transmission of items. (Recall that all items in this paper are 32 bits.) The results are given in Figure 3. Clearly, using Bloom filters becomes more advantageous as the number of inserted items increases. Although Bloom filters result in larger transmissions when few elements are transmitted, even uncompressed Bloom filters become more efficient when slightly more than 2000 elements are transmitted. As mentioned in Section 2, the Bloom filters would perform even better if the size of an item were larger, because the effectiveness of a Bloom filter does not depend on the size of the items in the set it represents.

The effectiveness of the Golomb-Rice compression is related to the Bloom filter's false positive rate. Larger false positive rates result in better compression because fewer hash functions are used, and so every element insertion adds fewer 1's to the filter. This results in fewer run lengths of 0's and better compression gains. At a 1% false positive rate, compressed Bloom filters are more efficient than sequentially transmitted elements at 1100 elements. The crossover at a 0.1% false

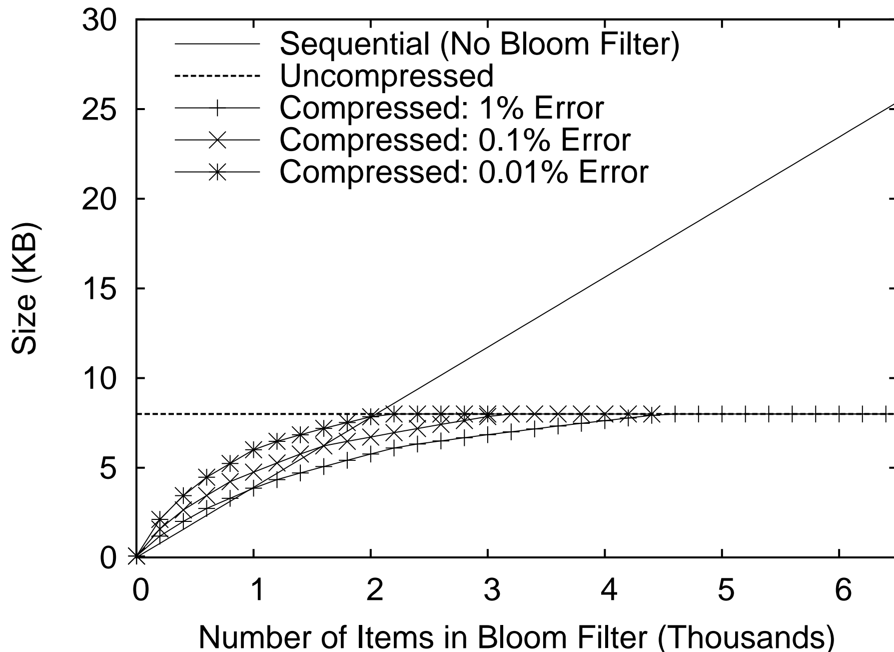

Figure 3: Comparison of transmission sizes for the sequential scheme and 8KB uncompressed and compressed Bloom filters.

positive rate occurs later at about 1600 elements and compression never reduces transmission size at a 0.01% false positive rate. Once again, these crossover points are essentially determined by the size of an item, and would naturally be smaller for larger items.

As mentioned in Section 4.3, the Golomb-Rice compressed Bloom filter is the same as the uncompressed filter when $M = 1$. Thus, when $M = 1$, we can skip the compression procedure entirely. This switch to uncompressed Bloom filters occurs when the Bloom filter is roughly 70% full. Indeed, one can derive this fact easily using the analysis in Section 2 and the fact that the numerically optimized value for p' where the Golomb-Rice parameter M changes from 2 to 1 is approximately $p' = 0.618$.

Compression affects the total amount of transmission in the network more than is initially apparent. Indeed, sparser Bloom filters will be more much common than near-capacity Bloom filters in the typical tree routing scheme. Although a few motes near the root of the network will send filters representing sets with many elements, the many more motes farther away from the root will likely have much sparser Bloom filters. Thus, we expect that in a typical application, a substantial fraction of motes in the tree will be in a position where the Golomb-Rice compression scheme gives a substantial reduction in transmission size over both the sequential and uncompressed Bloom filter methods.

Also note that uncompressed Bloom filters can be used to conserve memory when storing data locally using the distributed storage model in Section 4.1.2. When large numbers of elements require on-mote storage, Bloom filters provide an excellent way to increase effective storage by over 200%. Naturally, the storage improvement only grows with the size of the items.

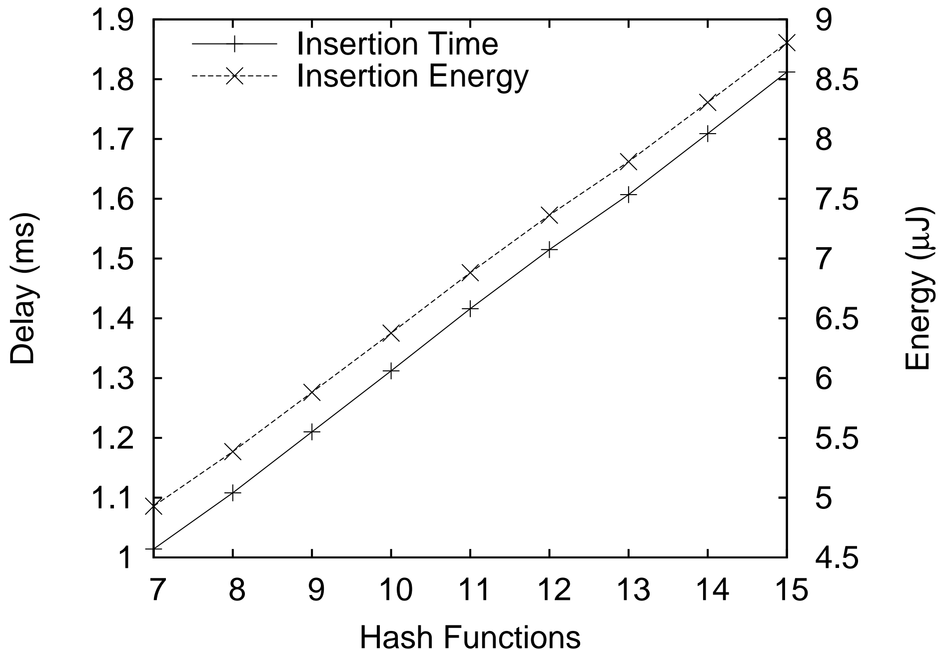

Figure 4: Estimated delay and energy requirements for inserting an element with various numbers of hash functions. Element insertion costs 0.100ms and 0.485μJ per hash function with an overhead of 0.912ms and 4.432μJ.

6.2 Timing

We now turn our attention to the time required to perform the various operations required by our system. In all cases, we find that the operations are certainly fast enough for the reference applications considered in Section 3.

We start with the time required to insert a single item. Naturally, this delay is roughly proportional to k, the number of hash functions used by the Bloom filter. Since k increases as the false positive rate decreases (for properly chosen Bloom filter configurations), insertions to filters with larger false positive rates require less time than insertions with lower false positive rates. As Figure 4 shows, configuration 1 in Table 1 ($f = 1\%$, $k = 7$) requires 44% less time per element insertion than configuration 4 ($f = 0.01\%$, $k = 15$). (There is an obvious correspondence between insertion time and energy cost, and so we plot them simultaneously in Figure 4 for future reference.) Since our hash functions are easy to evaluate, insertion is a computationally light task. In particular, about 1000 elements can be inserted per second in configuration 1.

Next, we consider the time required for the Golomb-Rice compression and merge operations. We show the results in Figures 5 and 6. Clearly, the times are suitable for the reference applications in Section 3. For example, consider a network of 100,000 motes arranged in a binary tree where each mote contributes the same number of elements to the Bloom filter. Then a back-of-the-envelope calculation shows that the additional delay due to Bloom filter processing is only 12 seconds over all hops in the worst case.

Also, it should come as no surprise that performance is related to the number of elements and the false positive rate of the filter. Indeed, the performance is largely dependent on the number of

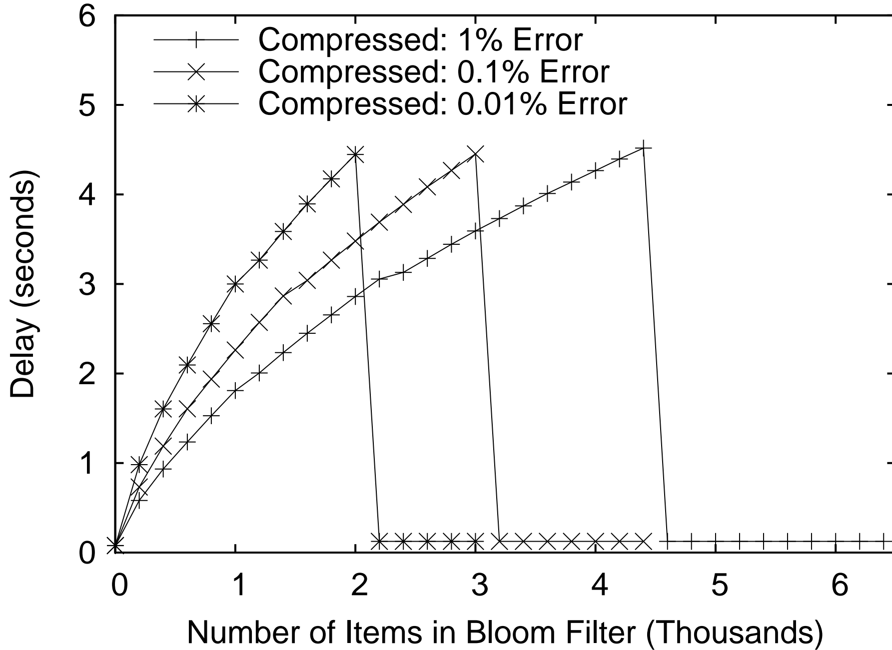

Figure 5: Time required to generate a compressed Bloom filter from uncompressed Bloom filters.

run lengths of 0's between 1's in the filter. As discussed in Section 6.1, either increasing the number of inserted elements or reducing the false positive probability (thus increasing the number k of hash functions) increases the number of 1's in the filter. As the number of 1's increases, so does the number of run lengths of 0's, and thus the time required to perform the Golomb-Rice compression and merge operations. We also note that, as explained in Section 6.1, uncompressed Bloom filters are always transmitted at approximately 70% capacity and beyond. In this range, the value of $M = 1$ is calculated and compression is determined unnecessary. As a result, the computation time drops immensely.

Finally, we turn our attention to the total time to send a set of elements over one mote-to-mote hop. This delay includes the time required to compress, transmit, and uncompress the Bloom filter when compressed Bloom filters are used and only transmission time when either uncompressed Bloom filters or sequential item transmissions are used. The results are shown in Figure 7. For the sake of completeness, we note that Figure 7 does not illustrate the amount of time required for just radio transmissions. Unsurprisingly, however, radio-only timing comparisons are similar to the transmission size comparisons in Section 6.1 (and Figure 3).

Returning to Figure 7, it is obvious that uncompressed Bloom filters can greatly reduce the total time when many elements are transmitted (at the cost of the false positive rate). Compressed Bloom filters clearly require more time than the sequential or uncompressed Bloom filter approaches. For this reason, compressed Bloom filters are not suitable for applications requiring low latency. In these cases, a combination of sequential and uncompressed Bloom filters should be used. However, compressed Bloom filters are still likely to be suitable for many instances of our reference applications, especially when energy considerations are taken into account; we show this in detail in Section 6.3.

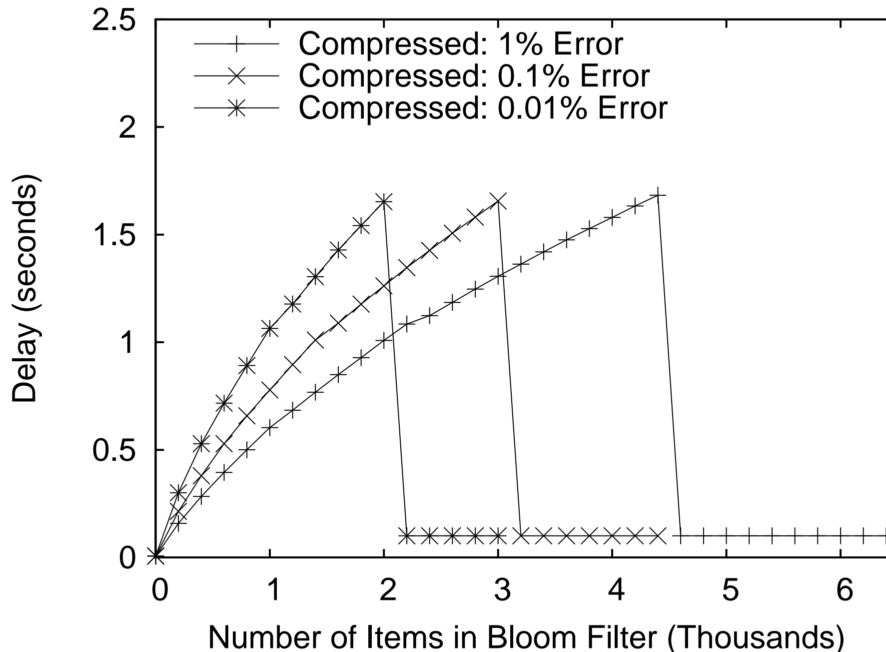

Figure 6: Time required to merge a compressed Bloom filter with an uncompressed Bloom filter in memory.

Furthermore, the compressed Bloom filter measurements in Figure 7 are made under worst-case assumptions. Specifically, that figure does not take into account the possibility for parallelism in performing compression operations simultaneously with radio transmissions. Instead, the figure assumes a pessimistic model where the compressed Bloom filter is transmitted in portions, and that we never transmit or receive a portion at the same time that we are compressing or decompressing a different portion. We present the figure in this way to emphasize that, even in this very conservative setting, our compression scheme is still practical.

Of course, it is useful to consider what happens when we allow transmission and compression operations to be performed in parallel. First, if we assume that transmissions handled by the radio do not simultaneously require CPU attention, then the delay introduced by compressions is reduced by up to 43%. Second, we may be able to parallelize the Bloom filter compression operations on the network level. Specifically, if each mote within communication range rotates radio control after every transmission, then each mote can perform compression operations while the others transmit. Since mote wireless transmission is half-duplex and only one mote in range can transmit at a time, but multiple motes can perform their own compression operations in parallel, this approach may be worthwhile.

6.3 Energy Consumption

We now give an overview of our system's energy usage. The energy cost of element insertion, shown in Figure 4, is naturally dependent on the number of hash functions used. Bloom filters with lower false positive rates require more hash functions and as a result use more energy. The energy cost is relatively low regardless of hash functions, on the order of μJ.

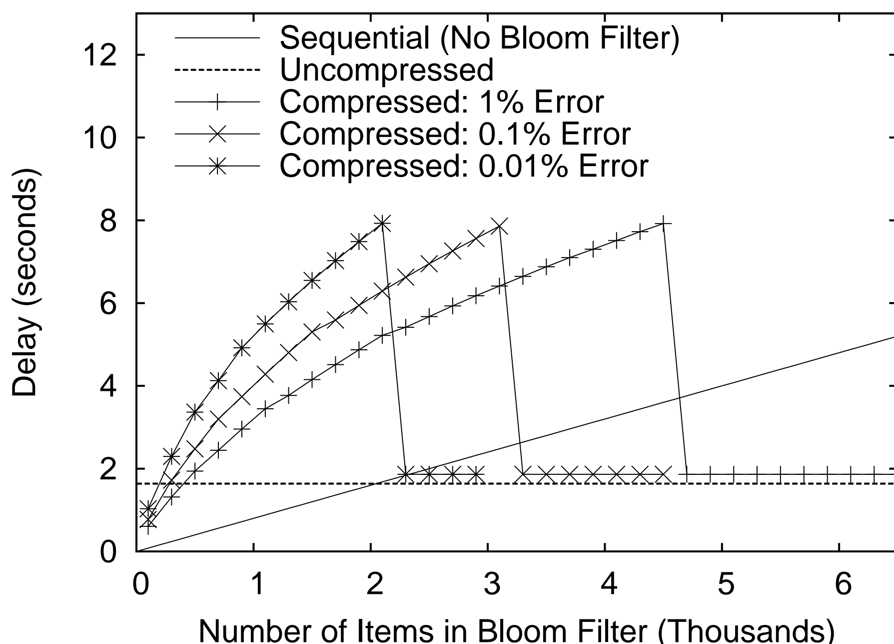

Figure 7: Total transmission hop delay including compression, transmission, and decompression delays where applicable.

The energy efficiency of our Bloom filter transmissions closely resembles transmission savings, as shown in Figure 8. This is not surprising, since the mote radio requires substantially more power than the processor and dominates system energy consumption. Although compressing Bloom filters requires extra computational energy, the reduction in transmission size typically saves considerably more energy. In all cases, the radio requires an estimated 97% to 99.9% of total system energy for transmissions. As a result, compressed and uncompressed Bloom filters can save up to 68% of transmission-related system energy. Since the motes that send the largest transmissions are the ones that experience the greatest energy savings, there is likely to be a substantial increase in network lifetime in a real application.

As mentioned previously, the theoretical effectiveness of a Bloom filter does not depend on the size of the items in the set it represents. Therefore we expect our system to perform even better for larger item sizes, when measured against sequential transmission of items. The one caveat here is that the complexity of the hash functions naturally increases with the size of the items, increasing the energy cost of element insertion. However, since energy usage is dominated by transmission sizes, we expect this effect to be negligible.

7 Conclusion

In this paper, we have discussed the use of Bloom filters to reduce energy and memory consumption in data intensive wireless sensor network applications. In particular, we have shown that our approach is feasible and worthwhile for currently available sensor network motes. We have also demonstrated the use of Golomb-Rice coding in sensor networks to achieve even smaller transmis-

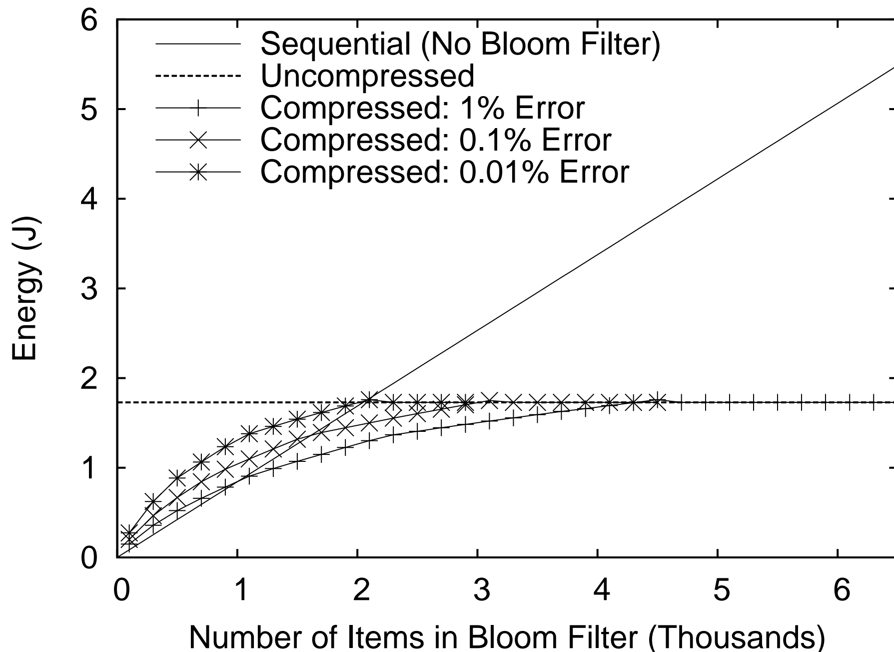

Figure 8: Total transmission hop energy cost including compression, transmission, and decompression energy costs where applicable.

sions and further reduce energy consumption. In summary, we have shown that our implementation is able to prolong sensor network life and simultaneously increase the amount of information recorded.

Acknowledgments

We are grateful to David Brooks and Michael Mitzenmacher for their comments and many useful discussions.

References

[1] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485-509, 2004.

[2] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18(2):143-154, 1979.

[3] B. Chen, K. Muniswamy-Reddy, M. Welsh. Ad-Hoc Multicast Routing on Resource-Limited Sensor Nodes. *Proceedings of the Second ACM/Sigmobile Workshop on Multi-hop Ad Hoc Networks: from theory to reality*, 2006.

[4] A. Ciancio, S. Pattem, A. Ortega, B. Krishnamachari. Energy-efficient data representation and routing for wireless sensor networks based on a distributed wavelet compression algorithm.

IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks, 309-316, 2006.

[5] J. Considine, F. Li, G. Kollios, J. Byers. Approximate Aggregation Techniques for Sensor Databases. *Proceedings of the 20th International Conference on Data Engineering*, 449, 2004.

[6] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., 1991.

[7] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19-51, 1997.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000.

[9] D. Ganesan, D. Estrin, J. Heidemann. Dimensions: why do we need a new data handling architecture for sensor networks? *ACM SIGCOMM Computer Communication Review*, 33(1):143-148, 2003.

[10] A. Ghose, J. Grossklags, J. Chuang. Resilient Data-Centric Storage in Wireless Ad-Hoc Sensor Networks. *Mobile Data Management - MDM 2003*, 45-62, 2003.

[11] G. Giorgetti, S. K. S. Gupta, G. Manes. Wireless localization using self-organizing maps. *Proceedings of the 6th international conference on Information processing in sensor networks*, 293-302, 2007.

[12] B. Greenstein, D. Estrin R. Govindan, S. Ratnasamy, S. Shenker. DIFS: a distributed index for features in sensor networks. *Proceedings of the First IEEE Sensor Network Protocols and Applications*, 163-173, 2003.

[13] P. Hebden and A. R. Pearce. Bloom filters for data aggregation and discovery: a hierarchical clustering approach. *Proceedings of the 2005 International Conference on Intelligent Sensors, Sensor Networks and Information Processing Conference, 2005*, 175-180, 2005.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister. System Architecture Directions for Networked Sensors. *Architectural Support for Programming Languages and Operating Systems*, 93-104, 2000.

[15] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. *Proceedings of the 6th international conference on Information processing in sensor networks*, 254-263, 2007.

[16] N. Kimura and S. Latifi. A Survey on Data Compression in Wireless Sensor Networks. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, 2(2):8-13, 2005.

[17] S. Madden, M. J. Franklin, J. Hellerstein, W. Hong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. *OSDI*, 2002.

[18] S. Madden, M. Franklin, J. M. Hellerstein, W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 491-502, 2003.

[19] D. Malan, T. Fulford-Jones, M. Welsh, S. Moulton. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. *Proceedings of the MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, 2004.

[20] J. Meany. Golomb Coding Notes, `http://ese.wustl.edu/class/fl06/ese578/GolombCodingNotes.pdf`, 2005. Accessed May 11, 2007.

[21] M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.

[22] Moteiv Corporation. Tmote Sky Datasheet, `http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf`. Accessed June 6, 2007.

[23] S. Nath, P. B. Gibbons, S. Seshan, Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 250-262, 2004.

[24] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237-1239, 1989.

[25] M. Rudafshani and S. Datta. Localization in wireless sensor networks. *Proceedings of the 6th international conference on Information processing in sensor networks*, 51-60, 2007.

[26] K. Sayood. *Data Compression*. Second Edition, Morgan Kaufmann Publishers, 2000.

[27] C. Taylor, A. Rahimi, J. Bachrach, H. Shrobe, A. Grue. Simultaneous localization, calibration, and tracking in an ad hoc sensor network. *The Fifth International Conference on Information Processing in Sensor Networks*, 27-33, 2006.

[28] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. *Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005*, 108-120, 2005.

[29] Wikipedia. Golomb coding, `http://en.wikipedia.org/w/index.php?title=Golomb_coding&oldid=119757295`. Accessed May 11, 2007.

[30] Wikipedia. Radio-frequency identification, `http://en.wikipedia.org/wiki/RFID`. Accessed June 11, 2007.