# Snoogle: A Search Engine for Pervasive Environments

Haodong Wang, Chiu C. Tan, and Qun Li

**Abstract**—Embedding small devices into everyday objects like toasters and coffee mugs creates a wireless network of objects. These embedded devices can contain a description of the underlying objects, or other user defined information. In this paper, we present Snoogle, a search engine for such a network. A user can query Snoogle to find a particular mobile object, or a list of objects that fit the description. Snoogle uses information retrieval techniques to index information and process user queries, and Bloom filters to reduce communication overhead. Security and privacy protections are also engineered into Snoogle to protect sensitive information. We have implemented a prototype of Snoogle using off-the-shelf sensor motes, and conducted extensive experiments to evaluate the system performance.

**Index Terms**—Snoogle, search, information retrieval, sensor networks.

## 1 Introduction

Wireless sensors have grown beyond motes scattered off a plane to collect environmental data. Sensors today can be found on diverse objects such as buildings, cars [1], and even clothing [2]. As sensors become more ubiquitous in our environment, their roles can extend beyond environmental sensing, to become an electronic representation of different objects. A sensor attached to a folder, for example, can contain a short summary of the contents of the folder. Information once scribbled onto post-it notes and stuck to the folder can also be stored directly onto the sensor itself. These sensors, which are considered as the representatives of the physical objects they are attached to, naturally form a database of the physical world. New techniques for searching information in such a database are necessary.

Information retrieval (IR) has been widely used to search for information[1] within databases. People can use search engines like Google, to easily find remote data objects. However, since the physical objects are disconnected from the cyberspace, searching for information in the physical world is more difficult. For example, a college student can easily search and view a Shakespeare manuscript on the Web using several mouse clicks, but may have to spend hours to find his notebook for an exam. This observation motivates us to develop an information retrieval system for the physical world.

A straightforward system design is to maintain a central database, and let each object return its location and data to this database. The user will query the database to find a particular object. However, since the data in an object can change, frequent updates to the database are needed. This poses a scalability issue when the number of objects increases, since database will be unable to support large numbers of simultaneous object updates. An alternative design of broadcasting a user query to all the objects instead of maintaining a database can eliminate the cost of frequent updates. Upon receiving a query, each object will determine whether its data match the query before deciding whether to respond. The user will collect the responses and determine the most suitable answer. However, the communication cost of delivering the query to all the objects is high when the number of objects is large. Furthermore, an object is unaware of the answers provided by its peers, and hence cannot accurately determine whether its own answer is best suited for the query. As a result, the user has to sift through a large number of responses to determine a suitable answer.

In this paper, we present Snoogle, an information retrieval system built on low-cost wireless sensor networks. In a pervasive computing environment, Snoogle serves as the search engine and helps people to search physical objects at their vicinity.

### 1.1 Challenges

While the use of IR in the Internet is well established, adopting IR within a sensor network poses several unique challenges. First, a sensor network has to limit communication to conserve power. Internet search engines can have spiders that continuously crawl the Internet for data. Large amounts of data can be collected and stored in a depository for further processing. Sensor networks do not have this luxury, and thus we cannot directly implement the data collection techniques for Internet search engines onto sensor networks. Novel data storage and collection techniques are necessary to overcome such limitations. Second, when we consider sensors being attached to physical objects, these sensors can be mobile and the stored data can change rapidly. Most web page locations, on the other hand, are comparatively more static even though the content could be very dynamic. This makes maintaining up-to-date information in a sensor network more challenging. Third, security and

1. Although information may be in many different types, this paper only focuses on the most popular textual information search.

- *H. Wang is with the Department of Mathematics and Computer Science, Virginia State University, PO Box 9068, 1 Hayden Drive, Petersburg, VA 23806. E-mail: hwang@vsu.edu.*
- *C.C. Tan and Q. Li are with the Computer Science Department, College of William & Mary, McGlothlin-Street Hall 126, Williamsburg, VA 23185. E-mail: {cct, liqun}@cs.wm.edu.*

*Manuscript received 14 Jan. 2009; revised 15 May 2009; accepted 10 Aug. 2009; published online 20 Aug. 2009.*
*Recommended for acceptance by C.-Z. Xu.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-01-0019.*
*Digital Object Identifier no. 10.1109/TPDS.2009.145.*

privacy are bigger challenges in the sensor network search than the Internet search. People may choose not to have a web page, or not to update it frequently. However, since sensors are attached to physical objects like clothes, a user may have several sensors they are not aware of. Furthermore, sensors always have fewer resources compared to web servers, making implementation of security even more challenging. In this paper, we focus on reducing the communication cost, and addressing security and privacy concerns.

### 1.2 Contributions

We have built Snoogle, an information retrieval system built on sensor networks for the physical world. Snoogle consists of three components, *object sensors*, *index points* ($IP$), and a *Key Index Point* ($KeyIP$), which is a super node that manages all $IP$s. These terms are defined later in the paper. We summarize our contributions below.

First, to the best of our knowledge, this is the first research work to propose and build an IR system for the physical world based on sensor networks. Existing IR systems have been deployed on large systems such as servers and desktop machine, and not on tiny, low cost, resource limited devices used in this paper. Prior work on sensor network data management investigated data query, or index building for sensor databases, but not IR.

Second, we examine compression techniques like Bloom filters and compressed Bloom filters to reduce the data transmitted within the sensor network. A Bloom filter is basically a way to compress information into a form that can still be searchable. It generates smaller amount of information for the objects to transmit to an $IP$, but incurs uncertainty about the original information, and thus might result in false positives during the search process (although no false negatives will appear). Our scheme tries to reduce as much information transmitted as possible, while reducing false positives. A false positive occurs when we cannot distinguish between two different terms that are mapped onto the Bloom filter bitmap. This work is interesting on its own merits since this opens another avenue for minimizing transmission costs in sensor networks. While bloom filters have been used in other systems, the combination of bloom filters into an IR based physical world search engine, which is built on top of sensor devices, is a new concept.

Third, we develop a distributed top-$k$ query algorithm to further reduce the communication cost for user distributed queries. Our theoretical analysis shows the message complexity is linear to the value of $k$ and the number of queried $IP$s, which is close to the optimal solution. Our simulation results are consistent with the theoretical analysis and demonstrate the significant message complexity advantage over the naive scheme. Unlike the top-$k$ algorithm proposed in prior work [3], our algorithm design addresses the challenges of message complexity problem on resource constrained sensor devices.

Fourth, we propose a more flexible security and privacy framework for a user to search a sensor network of objects. Each object defines its access attributes similar to a file in UNIX system, allowing for personal, group, and global permissions. A user must show that his access rights matches with the object's access attributes before searching an object. We use elliptic curve cryptography (ECC), an efficient public key cryptography (PKC) suitable for sensors, which has a clean interface with no messy key distribution and management phase, instead of the more common symmetric key cryptography used in sensor network research. Our proposed scheme is flexible enough to be easily adopted into other sensor applications. We are the first to integrate the resilient public key scheme into an access control security scheme for a practical sensor network application.

Fifth, we have built a sensor network information retrieval prototype on our mote testbed. Our prototype has integrated all the components, including IR, compression technology, distributed top-$k$ query scheme, and PKC-based access control on off-the-shelf sensor hardware. The prototype validates the approaches proposed in this paper, and provides experimental data on the indexing and searching performance. We also build a simulator to simulate a larger sensor network to demonstrate the scalability of Snoogle. Although the prototype built in this paper is not a mature system, the experience and experimental data collected in this work can serve as a guide for future full-fledged systems.

The rest of the paper is as follows: The next section presents the Snoogle system overview. Sections 3 and 4 examine the communication compression and query process, respectively. Section 5 explores the mobility and security support. Section 6 illustrates our prototype implementation, and Section 7 contains the evaluation results. We discuss our system limitation and related work in Sections 8 and 9. Finally, Section 10 concludes.

## 2 System Design

### 2.1 System Components

Snoogle consists of three main components: object sensors, $IP$s and $KeyIP$s. An object sensor is a mote attached to a physical object, and contains a textual description of the physical object. The object sensor can be either static or mobile, depending on whether the attached physical object is stationary or moving. Snoogle does not require object sensors to be homogeneous. Object sensors can be as powerful as an iMote [4] or MICAz mote [5], or as weak as an active RFID tag. Snoogle only requires all object sensors to communicate using the same radio frequency.

An $IP$ is a static sensor device that is associated with a physical location, for example, a specific room in an office building. $IP$s are responsible for collecting and maintaining the data from the object sensors in their vicinity. A typical $IP$ is battery powered, and equipped with a microcontroller, radio module, and a large amount of flash memory. A collection of $IP$s forms a homogeneous mesh sensor network.

The $KeyIP$ collects data from different $IP$s in the network. The $KeyIP$ is assumed to have access to a constant power source, powerful processing capacity, and possess considerable storage and processing capacity.

### 2.2 System Architecture

Snoogle adopts a two-tier hierarchical architecture shown in Fig. 1. The lower tier involves object sensors and $IP$s. Each $IP$ manages a certain area within its transmission range. Object sensors register themselves and transmit the object description metadata to the specific $IP$. $IP$s are responsible for building the inverted indexes for local search. We assume the object description data are either preloaded or incrementally uploaded by the object owner. For example, before a book is placed on the shelf for sale,

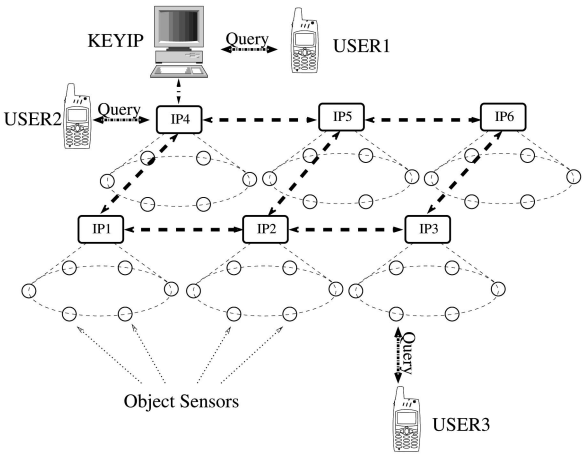



Fig. 1. Overview of sensors, $IP$s, and $KeyIP$ architecture.

the store staff attaches a sensor loaded with the book's introduction to the item. This is just like putting a price tag on the book. Later, the store staff can upload some reader reviews to the sensor attached so that the potential buyers can directly search the reviews from the book instead of via a remote website like *amazon.com*.

On the upper tier, $IP$s have dual roles. First, $IP$s forward the aggregated object information to the $KeyIP$ so that the $KeyIP$ can return a list of $IP$s that are most relevant to a certain user query. Second, $IP$s also routes traffic between $IP$s, the $KeyIP$, and objects. The $KeyIP$, considered as the sink of the network, holds the global object aggregation information reported by each $IP$. While Snoogle does not restrict the number of $KeyIP$s, we only consider a single $KeyIP$ setup in this paper for simplicity.

Users query Snoogle using portable devices such as smart phones or PDAs. Snoogle provides two different kinds of queries, a local query and a distributed query. A local query is performed when a user directs his query to a specific $IP$. This type of query occurs when a user wishes to limit his search to objects located at a specific location. A user performs a distributed query when he queries the $KeyIP$. The distributed query capability allows for scalability since users do not need to flood every $IP$ to find a particular object.

### 2.3 Data Processing in Object Sensors

Each object sensor contains two types of data, *payload data* and *metadata*. Payload is the short description about the particular physical object. Metadata is a representation of the payload data. For example, consider an object sensor attached to a folder. The payload data could be a short note describing the contents of the folder. The metadata is a set of tuples, $\{term_1 : freq_1 : id\} \cdots \{term_n : freq_n : id\}$, where $term$ is a single word describing the payload data, and $freq$ indicates the importance of this term in describing the payload data. A user storing information into an object sensor will create both the payload data and metadata. To minimize the data transmission cost, the data in the object sensor can also be precompressed using compression schemes described in the next section.

### 2.4 Data Processing and Storage at $IP$s

$IP$s in Snoogle have two data processing roles. First, $IP$s collect metadata from objects within their range and

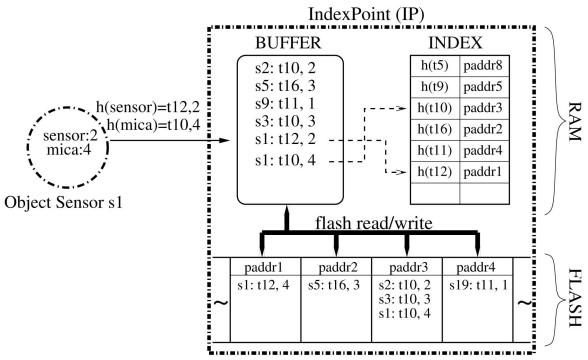



Fig. 2. Sensor S1 sending data to $IP$.

organize the data into indexed chains. Due to reliability and limited memory concerns, these chains are stored in the sensor's onboard flash memory rather than RAM. Second, $IP$s have to periodically send aggregated information to the $KeyIP$ so that the $KeyIP$ can maintain its inverted index of $IP$s information. $IP$s perform the following three data operations:

**Insert**. This operation is executed when a new object comes into the $IP$'s region and sends its metadata to the $IP$. The $IP$ stores the new metadata with the associated term frequencies and object ID into its inverted table.

**Delete**. When a physical object leaves the vicinity of a particular $IP$, e.g., a user moves a book from one office to another, the corresponding object is no longer associated with that $IP$. The $IP$ then performs a "delete" operation to remove all the metadata of the leaving object from the corresponding chains.

**Modify**. This operation is performed when there is a change in the object's data. When this happens, the object sensor sends a modification request to the $IP$. Since the corresponding chains are stored in the flash memory, which does not support random writes, the "modify" operation is achieved by the combination of a "delete" and an "insert."

We let the $IP$s only store the metadata of the objects, instead of the entire payload data in flash memory in order to conserve storage space. We take advantage of the small granularity write capability of the NOR flash (TelosB onboard flash memory) and allow $IP$s to be able to append the object metadata sequentially in the flash memory.

We also implement a "delete" function that efficiently invalidates the metadata associated with an object. We perform this "delete" by zeroing out the necessary bytes in the flash memory, avoiding the expensive *read and write* method used in general flash storage system. That same memory location is not overwritten until there is a sector delete during garbage collection.

After the object sends its ID and metadata to the $IP$, the information is first stored in a buffer in RAM. Fig. 2 illustrates the $IP$ storage architecture. Once the buffer is full, a hash function is applied to every term in the buffer. The hash results are used as the indices that map to the lookup table entries. We maintain the lookup table (INDEX in Fig. 2) in RAM to store the address pointing to the flash page. Each flash page has the size of 256 bytes. Those flash pages which are associated to the same lookup entry are organized in a chained structure, very similar to the structure of the linked list in data structure. The value of the lookup table entry always points to the head of the flash page chain. The most

populated terms that are mapped to the same lookup table entry are flushed to the flash memory, and the flash address is returned to the lookup table entry. This flushing operation continues until there are enough empty buffer slots to hold the incoming object terms. The lookup table manages the flash addresses in a chained structure that multiple flash pages can be assigned to the same table entry.

When an $IP$ receives a query, it applies the hash function to the query to map each query term to a lookup table entry, and obtains the flash address. This address stores a location of the flash page chain head which contains that particular term. Next, each flash page in the chain is sequentially read to the RAM, and scanned for the matching elements. Eventually, a list of matching terms with associated object IDs is obtained, then a ranked list of object IDs that best match the query is derived using an IR algorithm elaborated in the next section.

Finally, each $IP$ will periodically send the updated metadata terms and objects, which reflect the object dynamics in the region, to the $KeyIP$. The $KeyIP$ stores the data and checks for inconsistency. This inconsistency occurs when objects moved from one $IP$ to another before the $IP$s have a chance to update their information. Since all objects have their unique IDs, this inconsistency can be easily detected by the $KeyIP$. The $KeyIP$ then informs the involved $IP$s to verify the object data. For example, both $IP_1$ and $IP_2$ report having object $s_1$. Each $IP$ will send a message directed to $s_1$. If $s_1$ is no longer in the range of $IP_1$, then only $IP_2$ will receive a reply. $IP_1$ will flag $s_1$ as no longer present and inform $KeyIP$. The same holds if $s_1$ is no longer in the range of $IP_2$. If $s_1$ falls in the intersection of both $IP$s, $s_1$ will reply to both and $KeyIP$ is not updated.

### 2.5 Additional Discussion

When an object lies within the vicinity of multiple $IP$s, the object has to determine which $IP$ to select to transmit its data. Ideally, the nearest $IP$ in terms of the closest physical distance to the object is a good criteria. However, this may not be practical because the $IP$ deployment may be restricted at certain locations due to the physical limitations. In this paper, we use a predetermined mapping table to identify the $IP$ that the object sensor should select. The lookup table maps the RSSI pattern to a specific $IP$. In this scheme, the sensor will sample the RSSI values from multiple $IP$s, and query the nearby $IP$s for the designated $IP$s given the RSSI readings. For example, a first-aid kit placed in the cabinet should select the room $IP$ mounted on the other side of the wall rather than a closer $IP$ that is mounted in the next room. This lookup table can be precomputed ahead of time, and can be stored in $IP$ flash memory.

The use of RSSI for localization has been widely studied, and Snoogle can be modified to use more advanced localization algorithms [6], [7], [8], [9], [10] to achieve more accurate localization.

While we do not specify the maximum number of objects that can associate with an $IP$, we assume that there should not be more than about 200 objects that lie within the range of a single $IP$. The reason is that even though object sensors are small, the sensors are attached to larger physical objects like laptops and coffee mugs. For a smaller space like an office cubical, the number of tagged things are unlikely to be in the hundreds or thousands; thus, the $IP$ never has to index so many items. Furthermore, since the $IP$ will delete data from object sensors that have left its vicinity, there will be no accumulation of data.

Larger spaces such as a warehouse storage area may contain thousands of objects. In this situation, multiple $IP$s can be installed to index the data from the objects. As mentioned earlier, an object facing a choice of multiple $IP$s will send the RSSI values to the $IP$s which will then assign an $IP$ for that object to associate with. This way, the objects can be associated with an appropriate $IP$ to facilitate searching, and will not be concentrated into a single $IP$.

## 3 Communication Compression

A Bloom filter [11] is used in Snoogle to compress groups of terms together. A Bloom filter with an $m$-bit array and $k$ independent hash functions are used for every $n$ words. The $m$-bit array is first initialized to "0." Then, for each word, the hash function maps the input to a value between 0 and $m-1$, corresponding to the bit position in the bit array, and that bit is then set to "1." After $n$ words are inserted, the resulting value of the array becomes the summary of the $n$ words. The collection of the arrays becomes the summary of the document. To check whether or not a word is in the document, we apply the $k$ hash functions to the word and check if the resulting bit positions are all "1"s in any of the array collection. A single "0" indicates there is no match. However, the result of all matching "1"s only indicates there is a certain probability that there is a real match. The uncertainty is due to false positive or a collision. We use the terms false positive and collision interchangeably in this paper. If a Bloom filter has $m$ bits, $k$ functions, and holds $n$ words, the probability of having a collision (incurs the false positive) with another word is

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx (1-e^{-kn/m})^{k}. \tag{1}$$

When $m$ and $n$ are fixed, the optimal false positive rate can be achievable when [12]

$$k = ln2 \cdot \frac{m}{n}. \tag{2}$$

Bloom filters can be further compressed to achieve better transmission efficiency [13]. This is based on the observation that an $m$-bit string may be transmitted by a less number of bits without any information loss. We denote $z$ as the number of bits after compression. Note that the compression only works ($z < m$) when there are less "1"s than "0"s (or in reversed case). Mitzenmacher [13] indicated that each bit of the Bloom filter has roughly $1/2$ probability to be "1" or "0" when a Bloom filter is tuned to have optimal false positive rate. This suggests that an optimal Bloom filter almost cannot be compressed. It also means that there is trade-off between false positive and compression ratio. To gain transmission efficiency, we have to sacrifice the false positive rate. Mitzenmacher [13] also noted that the procedure of compressing a Bloom filter is actually equivalent to hashing each term into a $z/n$ bit string. Therefore, instead of doing complicated bit operations, we simply hash each term to a $z/n$ bit string, and concatenate the $n$ hash results together to generate an array. Assuming that the hash function is perfect, the probability of having a collision with another word for each $z/n$ bit string is roughly $(\frac{1}{2})^{z/n}$.

Selecting the correct compression method is crucial for Snoogle system. The optimal bloom filter achieves the lowest false positive rate, while the compressed bloom filter scores better compression ratio [14] so that it can achieve better transmission efficiency and lower processing overhead. We believe that the low transmission cost and processing overhead are more desirable for extremely resource constrained sensor nodes. Therefore, we use the compressed Bloom filter for our Snoogle system. In practice, with carefully chosen parameters, we can lower the false positive rate to an acceptable level. As we will describe in Section 7, given the data set with 1,512 words, and compressed Bloom filter size of 16 bits, the false positive rate is only about 2.3 percent.

## 4 Performing Query

In this section, we present the details and theoretical discussion of the user query process and the top-k query schemes.

### 4.1 Query Process

There are two ways of querying Snoogle. The first is to query an $IP$ directly, the second is to query the $KeyIP$ first, and then to perform the distributed query given a list of most relevant $IP$s returned by the $KeyIP$.

The first query method is used when a user is only interested in finding the object in some specific area, or when the user has an approximate idea where the object might be found. For example, a user may want to find a magazine, but only if it is within a short distance from where he is currently located. Thus, he only queries the $IP$ nearest to him by sending a few terms that describe this magazine. The $IP$ then evaluates the query and returns an answer to the user. Each answer is the object ID that best matches the user query. The user can then query the object directly, or physically find the object.

The second query method is used when a user wishes to find an object regardless of where it is, or has no idea which $IP$ to start querying. The user first queries the $KeyIP$ with several terms describing the target object. The $KeyIP$ then returns a ranked list of $m$ $IP$s that contain objects that best match the query, where $m$ is a system parameter. The user then performs a distributed top-$k$ query from the returned list of $m$ $IP$s to find a satisfactory answer.

### 4.2 Scoring the Object Relevancy

When a user queries an $IP$, he receives a ranked list of object IDs that best match his query from the $IP$ as his answer. This ranking is derived from a score for each object contained within that $IP$ based on the query terms. For example, the user issues a query with two query terms, $(t_x, t_y)$ to an $IP$ with three objects, $(s_1, s_2, s_3)$. The score for $s_1$ is the sum of the weight of $t_x$ in $s_1$ and weight of $t_y$ in $s_1$. The score for $s_2$ and $s_3$ are determined in a similar fashion.

The weight of a term in a object is determined using the $TF/IDF$ weighing algorithm used in IR research. The intuition behind $TF/IDF$ is that the importance of a term in describing an object is based on two considerations. The first is the number of times that term appears in that object description, the $TF$. The greater number of times a term appears, the more relevant that term is in describing that object. In our system, the $TF$ value is given as part of the metadata of the object.

The second consideration is how important that term is among the *collection* of all objects in a particular $IP$. The $IDF$ is determined as

$$IDF = \log\left(\frac{\text{Total number of objects}}{\text{Number of objects containing the term}}\right).$$

The idea here is that if a term appears in many objects found within an $IP$'s neighborhood, it is less important. Consider an extreme case that a term appears in every object under an $IP$. Then, any one of the objects returned will contain that term, making that term not descriptive of any one object at all. To device the $IDF$ value, we need the total number of objects, and the number of objects containing the term. The total number of objects is easily obtained since an $IP$ knows all the objects in its neighborhood. The number of objects with a term is determined while processing the query at an $IP$. For a query term, an $IP$ counts the number of the matches with stored terms in its flash memory.

Putting it all together, the weight for a term $t_x$ in an object $s_1$ is

$$\text{Weight of } t_x = (TF_{t_x} \text{ in } s_1) \cdot (IDF_{t_x} \text{ in } s_1).$$

The above $TF/IDF$ scoring methods can also be used to evaluate the weight of $IP$s and objects in the distributed query. If object scores from different $IP$s are compared with each other in a distributed query, the $IDF$ values used have to either be normalized [15] or be replaced by global $IDF$s due to the object variation among the $IP$s. A discussion of $IDF$ normalization in top-$k$ query schemes is given in the next section.

The use of the $IDF$ allows the appropriate answer to be derived when comparing different $IP$s. Consider for a system instance with just two $IP$s, $IP_a$ and $IP_b$, where $IP_a$ is placed at a music CD store, and $IP_b$ is placed inside a student's dorm room (we assume the object population in the music store is much larger than those in the dorm). When the student wants to query for his **own** CD, he would like to obtain an answer from $IP_b$ rather than $IP_a$. However, since $IP_a$ is placed in a store with a lot of music albums, $IP_a$ will contain more terms associated with music albums, even though the appropriate answer should be from $IP_b$.

If our scoring algorithm only used the $TF$, then the score from $IP_a$ will be better than $IP_b$, since there are more objects in $IP_a$ that contain album terms. However, using $IDF$ (the global $IDF$ for the two $IP$s) means that the scores from $IP_a$ will be smaller since the album terms appear in almost all the objects in $IP_a$, resulting in a much lower overall score. This behavior allows $IP_b$ to be returned to the student as the most likely location for his own CD album.

We initially considered CORI weighing algorithm [16] when a user queries the $KeyIP$, but there was no noticeable improvement. Thus, we use a simple $TF/IDF$ algorithm throughout this paper.

### 4.3 Performing Top-$k$ Query

While Snoogle is capable of returning a ranked list of all relevant objects matching a query to a user, a user will usually want to limit the number of replies due to limited device display or battery power. Snoogle allows the user to specify a top-$k$ query which returns the $k$ best matches to a user query. The $k$ is a user specified value.

For a local query, returning the top-$k$ query is straightforward since an $IP$ needs to only return the top $k$ answers to the user. For a distributed query, a naive method is for the user to perform a top-$k$ query for each of the $m$ $IP$s returned by $KeyIP$. By collecting the $m \cdot k$ answers, the user can then obtain the top $k$ objects. However, the message complexity of $O(mk)$ is too expensive for the energy constrained system.

Our distributed top-$k$ query algorithm is shown in Algorithm 1. The intuition for the algorithm is as follows: Upon receiving a list of $m$ ranked $IP$s, the $KeyIP$ queries each $IP$ for the most relevant object, denoted as $ta_i, 1 \leq i \leq m$. The $KeyIP$ stores the $m$ objects in an array $a$ such that $a[i].obj = ta_i, a[i].weight = weight(ta_i), a[i].ip = IP_i$, where $weight(ta_i)$ returns the weight score determined by $TF$ and $IDF$ as we discussed previously. After collecting the highest weighing objects from all $m$ $IP$s, the $KeyIP$ sorts the objects in the descending order of the object weight, and obtains a new array that $a[1].weight \geq a[2].weight \geq \cdots \geq a[m].weight$. The first top-$k$ answer, $a[1].obj$, is immediately available. The $KeyIP$ sets the threshold value as $a[2].weight$, and queries $a[1].ip$ for the objects (excluding $a[1].obj$) that weights more than the threshold value. Note that among all the $m$ $IP$s, it is possible for $IP$ $a[1].ip$ to solely hold objects with weights larger than $a[2].weight$, so there is no reason to first query the other $IP$s. Ignoring objects that are designated as top-$k$ objects, each $IP$ has a new highest weighing object, and the same process continues till all top-$k$ objects are found. The algorithm stops any time when $k$ top objects are retrieved, and the $KeyIP$ returns the answer to the user.

**Algorithm 1.** Distributed Top-$k$ Query Algorithm

1: Input: $k$ $IP$s: $IP_1, IP_2, \ldots, IP_m$
2: Output: top-$k$ answers: $Obj_1, Obj_2, \ldots, Obj_k$
3: Each $IP$ sorts its objects in descending order of the weights
4: **for** from $i = 1$ to $i = m$ **do**
5: query $IP_i$ for the top answer; each $IP$ removes the first object from the sorted list and sends it to user
6: store the top answer $ta_i$ and its associated weight in an array: $a[i].obj = ta_i, a[i].weight = weight(ta_i)$, $a[i].ip = IP_i$
7: **end for**
8: set the number of committed objects, num_commit=1
9: **while** num_commit $< k$ **do**
10: sort the array in descending order of weight so that $a[1].weight \geq a[2].weight \geq \cdots a[m].weight$
11: send $a[2].weight$ and num_commit to $IP$ $a[1].ip$
12: IP $a[1].ip$ removes from its sorted list a list of objects (say $l$ of them) such that the last object has the highest weight less than $a[2].weight$, say $w$
13: IP $a[1].ip$ sends the first min(l, $k -$ num_commit)
14: commit all retrieved objects with weight greater than $a[1].weight$, change the value of num_commit, set $a[1].weight = w$
15: **end while**
16: return all the committed objects $Obj_1, Obj_2, \ldots, Obj_k$

An important issue in the above algorithm is the accuracy of the merged object ranking. Note that the object weights reported by each $IP$ are local scores, which are determined by local $IDF$s. However, the local weights cannot be compared directly [15]. We consider following two solutions: 1) normalizing the local scores; 2) calculate the global scores. As indicated in [15], the common normalization scheme requires the exchange of object statistics among $IP$s, which may incur large amount of communications between $IP$s. On the other hand, the heuristics used in the proposed normalized score estimation are tied to the specific database and, therefore, cannot be used in Snoogle. In this paper, we choose to calculate the global $IDF$. Upon receipt of a user query, the $KeyIP$ first query $m$ $IP$s for the local $DF$ value of each term. After collecting all local $DF$s, $KeyIP$ immediately computes the global $IDF$s and sends them back to $IP$s. From now on, the weights computed at each $IP$ becomes global scores and then can be compared with each other. Although this approach requires an extra round of communication between $KeyIP$ and $IP$s, the actual cost is bounded by $2m$, where $m$ is the number of $IP$s.

To bound the number of messages transmitted in the process, we make the following observations: First, each $IP$ transmits at most one object that will not appear in the top-$k$ list. Therefore, the number of messages sent by all the $IP$s is at most $m + k$ including the top-$k$ objects and other objects that will not appear in the top-$k$ list. Second, for each query sent out to the $IP$, we will get back at least one object (which may appear or not appear in the final top-$k$ objects). Thus, the number of queries sent out to all the $IP$s is bounded by the number of received objects, which is at most $m + k$. From these two observations, the number of messages in this process is at most $2(m + k)$, which is more efficient than $m \cdot k$ in the naive scheme.

## 5 Mobility and Security Support

Here, we present the system enhancement for supporting the mobile objects, and describe a flexible and resilient security mechanism for private objects.

### 5.1 Supporting Mobile Objects

An $IP$ needs to keep up-to-date information about the object sensors in its neighborhood. However, since objects can be mobile, there will inevitably be object sensors moving in and out of an $IP$'s neighborhood. Snoogle uses a combination of *beacon* and *timer* methods to maintain updated information.

In the *beacon* method, the $IP$ will periodically broadcast a beacon that identifies itself. An object sensor in the neighborhood that receives this beacon will compare it against the previous beacon. If both beacons match, this indicates that the physical object's metadata has already been sent to that $IP$, and the sensor does nothing. Otherwise, it indicates that the physical object has moved to a new location, and sensor will send the metadata and ID to the new $IP$.

In the *timer* method, the communication is initiated by each individual sensor. Each object sensor periodically broadcasts a "keepalive" message. At the same time, the $IP$ maintains a timer. If the $IP$ does not receive any "keepalive" message from a certain associated object before the timer expires, the $IP$ considers the object to have moved away, and then deletes all the data of the object sensor from its storage. The *beacon* and *timer* methods can vividly regarded as "pull" and "push." In the *beacon* method, $IP$s pull the status information from the object sensors. In the timer method, object sensors push their status to $IP$s.

TABLE 1
Summary of Snoogle Implementation

| $KeyIP$ | Laptop Computer |
|---|---|
| Index Point | TelosB mote<br>Code size (binary): 31.7K bytes<br>Data size: 3475 bytes<br>Index table: 16 entries<br>Buffer size: 64 metadata<br>Flash page size: 256 bytes |
| Object Sensor | TelosB mote<br>Code size (binary): 19.3K bytes<br>Data size: 6054 bytes |
| User Module | TelosB + iPAQ<br>Code size (Java class):10.8KB |

The *beacon* scheme consumes less energy than the *timer* method. The object sensors only need to wake up in the duty cycle to listen the beacons. They do not need to transmit any message as long as there is no movement. The *timer* method, however, offers better reliability. When an object moves to another $IP$ neighborhood, the previous $IP$ can notice an object missing through the timer, and the new $IP$ also can also be notified by the timer message sent by the moving object. In short, the *beacon* method is more suitable for static objects, while the *timer* method works better for mobile ones. In practice, the two methods can be properly combined depending on the system requirement.

### 5.2 Providing Security and Privacy

Since Snoogle is built on sensors, Snoogle shares the same security threats as other sensor network applications. Furthermore, Snoogle also poses unique security and privacy requirements due to the search function which may violate personal privacy by revealing object information to others. For example, a user may not want his private object (i.e., DVD movie) to be searchable by strangers, but only his friends and himself.

Based on the above concerns, Snoogle must have a security mechanism to prevent objects from being searched by unauthorized users. We adopt WM-ECC [17], an efficient ECC suite customized for sensor devices as the security primitive. Prior work [18] has indicated that the public key scheme provides a cleaner user interface and outperforms the symmetric key schemes in memory overhead and energy consumption. The reason we choose ECC over more popular RSA is that ECC can be more efficiently implemented in resource constrained sensors. On TelosB sensor motes, it takes 1.4 s to generate a public key. To the best of our knowledge, this is the best ECC performance achieved among the academic implementations. In Snoogle, the access control is performed at the $IP$ instead of at $KeyIP$ in a distributed fashion.

We provide security for Snoogle by adding a security tag field to the object sensor. The security tag has an *OwnerID* field and a *GroupMask* field. The *OwnerID* refers to the owner identification. The *GroupMask* determines which group of users has the privilege to access the object. The ECC-based user authentication is very similar to RSA. If a user wants to search private objects, he first sends the query and the certificate, where the certificate is issued by Certification Authority, which can be Snoogle administration. The $IP$ first verifies the user certificate and then makes sure that the corresponding *OwnerID* and *GroupMask* match with the object tag. In the next step, the $IP$ uses the user public key derived from the certificate to encrypt a randomly chosen secret key, and sends the ciphertext to the user. If the user can successfully decrypt the key, it proves that the user is the legitimate owner of the certificate. Finally, the key is used to establish the secure channel between the $IP$ and the user. This key can also be used to achieve the user privacy, since the user can simply encrypt his query terms by using the key so that no one can learn the query content.

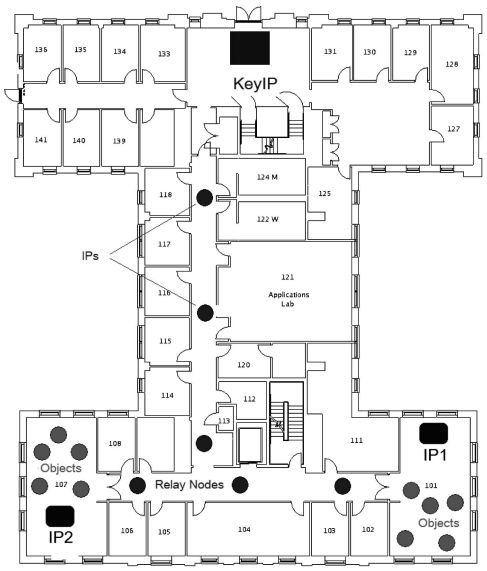



Fig. 3. Floorplan for testbed. The $KeyIP$ represented by a rectangular box is placed in the lobby. The two $IP$s, $IP1$ and $IP2$, also represented by a rectangular box, are located in two wings of the building, in rooms 101 and 107. There are five object sensors (squares) in the neighborhood of each $IP$. To route the messages between $IP$s and the $KeyIP$, we also deploy six IPs (round dots) in the hallway.

## 6 Prototype Experience

### 6.1 System Setup and Parameters

We implement a prototype of Snoogle, including object sensors, $IP$s, $KeyIP$, and user module, on TelosB motes, a research platform developed by Berkeley. TelosB hardware features a lower power TI MSP430 16-bit microcontroller with 10 KB RAM and 48 KB ROM. The onboard IEEE 802.15.4/ZigBee compliant radio transceiver facilitates the wireless communication with other IEEE 802.15.4 compliant devices. TelosB also has an onboard flash memory with 1 MB space, which enables our prototype $IP$ to store as many as 262,144 terms and the associated object IDs and term frequency. The low-power feature ($5.1\ \mu$A current draw in sleep mode) of TelosB motes allows object sensors to stay alive for long time. We use an HP iPAQ for the user module. The HP iPAQ features a 522 MHz ARM920T PXA270 processor, 64 MB RAM, and 128 MB flash memory. The software of $IP$s, object sensors, and user module are written by NesC language on TinyOS version 1.1.15. Table 1 illustrates the summary of the implementation.

We adopt the RC5 block cipher as the cryptographic hash function used to implement the compressed Bloom filter. We choose 16-bit as the Bloom filter size. Given our data set with 1,455 unique terms, the false positive rate is only 2.3 percent.

We set up a Snoogle network in our computer science building. The floorplan and the deployment of object sensors, $IP$s, and the $KeyIP$ are shown in Fig. 3. Our

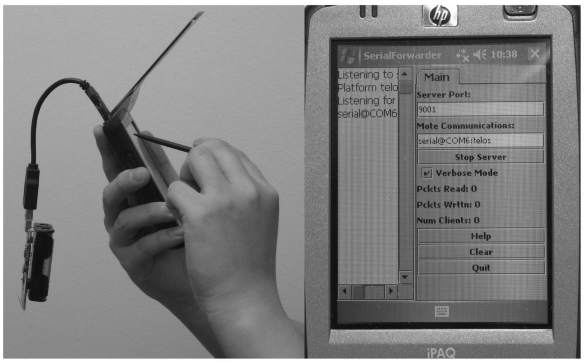



Fig. 4. Picture of a $PDA$ Query Device.

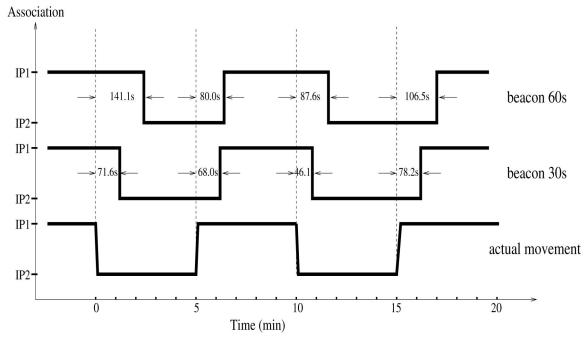



Fig. 5. Mobility test with beacon method.

experiment consists of two $IP$s located in the two wings of the building, in rooms 101 and 107. We attach a laptop to each $IP$ to collect timing information. The laptop plays no role in processing any of the data sent from sensor to $IP$. Each $IP$ has in its neighborhood five sensors which simulate sensors attached to different objects. Each sensor contains data from one complete conference paper. Details of the sensor workload will be discussed in Section 7. The $KeyIP$ is placed in the lobby, and consists of a laptop using a TelosB sensor as a communications module. Since the $KeyIP$ in our scheme can be a server, all data processing is performed by the laptop. We deploy several additional $IP$s to route data to the $KeyIP$.

We assume that a user will query Snoogle with a PDA like iPAQ. Since iPAQ does not directly support sensor communications, we use a TelosB mote to attach to the iPAQ through a USB adaptor as the front-end radio communication module. The iPAQ is running Windows Mobile 5.0 and Mysaifu JVM [19]. Fig. 4 shows a picture of our PDA querying device.

### 6.2 Prototype Test

We use the prototype tests to demonstrate the validity of the Snoogle architecture in a real world environment. The evaluation of specific Snoogle components is left to the next section. We consider the following two tests: First, can a user successfully query the $KeyIP$ to get a list of $IP$s, and can he query an $IP$ to obtain a list of sensors? Second, can the $IP$ and the $KeyIP$ effectively and accurately detect and manage mobile sensors?

Our first test emulates a user's query experience. We are particularly interested in the time duration for a user to get the object he searches for. A graduate student wants to search an academic publication. He first enters the lobby of Computer Science building, and uses his iPAQ to query the $KeyIP$ with the paper keywords. The $KeyIP$ immediately replies him with the list of $IP$s that carries the record as well as the associated term frequency information. Given the information replied from the $KeyIP$, the student picks the $IP$ which most probably contains the paper he is looking for, which is $IP1$ in our experiment setup. He then immediately queries $IP1$ again. This time, $IP1$ gives more detailed answers which finally help the student to find the paper. The time duration for the whole procedure is 1 minute and 45 seconds. Most of the time is spent on walking across the hallway and operating iPAQ. The $KeyIP$ and $IP$ query response time contributes a very small portion of the total time consumption. It only takes 41 and 55 ms for the $KeyIP$ and the $IP$ to reply the user query, respectively.

Our second experiment tests whether or not the $KeyIP$ and $IP$s are able to give a correct answer to the query for a mobile object. We implement both the beacon and timer methods in the prototype and test them separately. In each test, we set the cycle period with 30 and 60 seconds. The mobile object starts at the neighborhood of $IP1$. When the experiments begin, one of our group members takes the mobile object and walks cross the hallway to room 107. He stays in room 107 for 5 minutes and then carries the object back to room 101. During this period, we keep track of the object status in the $KeyIP$.

The test results of the beacon method is shown in Fig. 5. In the test with 30 s beacon cycle, it takes 71.6 s for the $KeyIP$ to get the object update report from $IP2$. Once the update arrives, the $KeyIP$ detects the object was originally associated with $IP1$. The $KeyIP$, therefore, immediately issues a notification to $IP1$. Note $IP1$ has no way to notice the missing object in the beacon method. After 5 minutes, the object returns to room 101, it takes 68.0 s for the $KeyIP$ to receive the update from $IP1$. Similarly, the $KeyIP$ issues a notification to $IP2$. When the beacon cycle is extended from 30 to 60 s, it takes longer time to detect the associate change.

Fig. 6 plots the mobile object association changes at the $KeyIP$ in the timer method. Comparing the two methods, the timer method is more reliable than the beacon method. The timer method allows both the $IP$ and the object to detect the movement while the $IP$ cannot detect the leaving object in the beacon method. That explains why it takes more time for the beacon method to detect the object movement once the object itself misses a couple of beacons, as shown in Fig. 5. However, the timer method requires

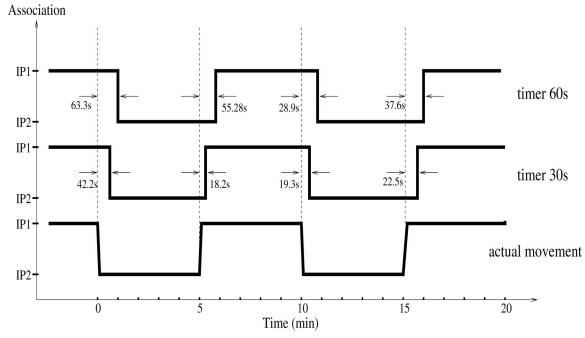



Fig. 6. Mobility test with timer method.

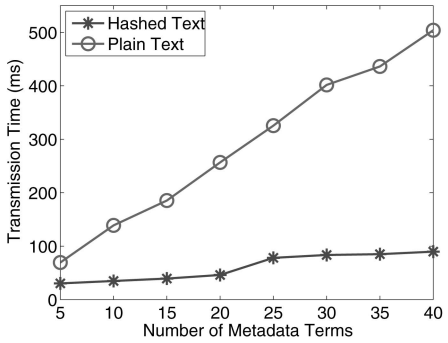



Fig. 7. Time taken to transmit metadata to $IP$.

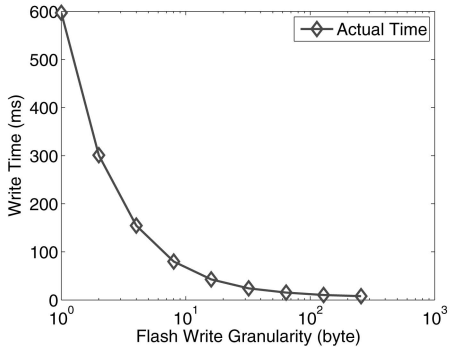



Fig. 8. The time delay to write 256 bytes of data with different write granularity.

more energy consumption for the object sensors because they have to keep sending "keepalive" messages. This is a disadvantage for energy constrained sensors. In practice, the two methods may be combined together to achieve the tracking performance to the energy consumption ratio.

## 7 Performance Evaluation

To better discern the performance of the system, we break the search system down into individual components and evaluate each component separately. We mainly focus on object sensor and $IP$ interaction because both the object sensor and $IP$ are power constrained and computationally challenged devices, while the $KeyIP$ is a resource-rich device. This makes the performance of the object sensors and $IP$s crucial for the validity of the system.

### 7.1 Workload Design

We use data from an academic conference to create our data set. The title, authors, and affiliations of each entry become the metadata terms in each object. We use the IR definition of $TF$ to obtain the weight of each metadata term. This yields a workload of 1,455 terms, which are sufficient for about 80 objects, each of which has about 15-25 unique words on the average. The average term size is between 7 and 8 characters. We further divide the 80 object sensors into eight $IP$s as the testbed for the distributed query.

### 7.2 Data Input and Maintenance at $IP$s

The startup phase for our search system occurs when the $IP$ is first initialized and contains no object data at all. This is a costly activity since the $IP$ has to identify all the objects within its range, and obtain their metadata. Fortunately, this initialization phase occurs rarely since the $IP$ utilizes persistent flash memory for data storage to protect against data loss. The main metric we use to evaluate this portion is the time latency needed for an $IP$ to obtain necessary data from object sensors and update the collected data for the future changes to give accurate answers for queries.

To reduce the transmission cost and improve the storage efficiency, Snoogle adopts the idea of compressed Bloom filter to compress the metadata terms. In particular, a hash function residing in the object sensor convert each plaintext metadata term into a 2-byte digest before transmitting the data over to the $IP$. We perform a comparison test to learn the benefit of the data compression. Fig. 7 shows the time taken to transmit hashed data to the $IP$ compared to the plaintext method. As we can see, the transmission time grows linearly as the number of terms increases when the plaintext data is used, while it takes much less time for the $IP$ to collect the same amount of data in the compressed form. It only takes 90 ms to collect 40 compressed terms. However, it requires more than five times of amount of time to transfer 40 uncompressed terms.

Note that the time taken to transmit hash terms is not proportional to the number of terms. For instance, the transmission time for five hashed terms is 30 ms, while the transmission time for 40 hashed terms is 90 ms. It only takes triple the amount of time to transmit eight times the data. The reason is due to TinyOS message overhead. For the communications between each object and an $IP$, we set the message payload size up to 60 bytes. Besides the 60-byte payload, each messages has 8-byte message header and 10-byte TinyOS header. We need three bytes to transmit one hashed text: one for the term frequency and other two for hashed value. Given a 60-byte payload, one message can carry up to 20 terms. Due to the above reason, even though the text size of 40 terms is eight times of that for 5 terms, it only takes one more message to transmit 40 terms, compared to that for sending 5 terms. Therefore, the transmission time for 40 terms should not be eight times of that for 5 terms.

Next, we show how the buffer helps to further improve the data collection efficiency. An $IP$ has limited RAM and uses flash memory to store the sensor metadata. The flash memory operation principle determines that the write in flash memory, specially in small granularity, is slow. We have performed a simple test to show the write efficiency at different write granularity. Fig. 8 plots the experiment result. As we can see, it consumes almost 0.6 s to write 256 bytes in flash with 1 byte write granularity, compared to 8 ms to write the same amount of data with 256 byte granularity. A common way to amortize the memory write overhead is to use a buffer. In Snoogle, each $IP$ maintains a small buffer in RAM of 256 bytes, to buffer object data before flushing to flash. Even though the NOR flash in TelosB supports random writes, we adopt the buffering approach to improve the efficiency. When there are multiple objects wanting to send data to an $IP$, the $IP$ will have to periodically halt transmission to flush the coming data into flash. The $IP$ does not need to invoke the expensive flash flushing routine as long as there is enough buffer space to hold the coming object terms, and picks a spare time later to flush the buffered terms into the flash.

Again, we conduct the comparison test to compare the two schemes. For the test case with an $IP$ buffer, we choose the buffer size with 256 bytes, equivalent to the page size of the

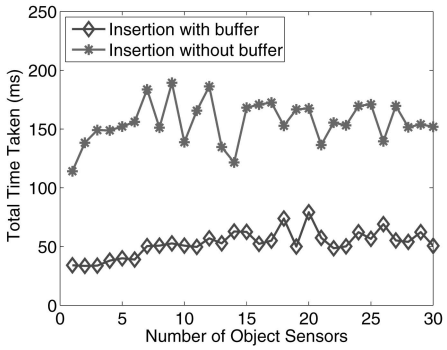



Fig. 9. Insertion performance with buffer and without buffer at $IP$.

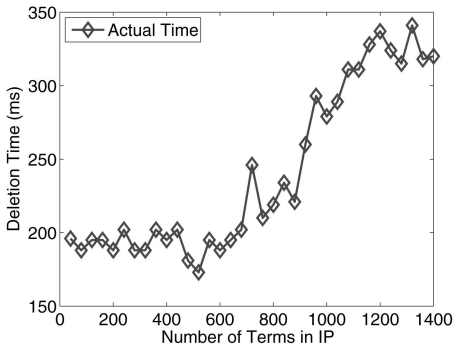



Fig. 10. The amount of time to delete an object with 10 terms.

flash memory setup. Since each object term requires 4-byte memory space, including 2-byte digest, 1-byte term frequency, and 1 byte for the object id, a 256-byte buffer can hold at most 64 object terms. In both the experiments, 30 object sensors, each having 10 terms, sequentially transmit the data to the $IP$. We record the average waiting time of each object sensor and present the results in Fig. 9. It clearly shows that each object sensor waits significantly less amount of time when the $IP$ uses the buffer.

We further notice that the variation of the object sensor waiting time without an $IP$ buffer is much larger. Our investigation reveals that the variation is determined by the amount of time taken to flush the data to the flash. Since each compressed term is further hashed by the $IP$ (as previously described in Section 4) to an index table, different terms can be mapped to different positions of index entries. The number of entries can be any value between 1 and the number of terms. The bigger the number is, the longer time is required because the $IP$ has to flush more flash pages. As the comparison, this variation is much smaller with a buffer enabled $IP$. The reason is that the $IP$ buffer keeps track of the index entry position of each term. When the number of buffer empty slots is not enough to hold the coming data, the buffer first flushes the most populated terms that hashed to the same index position, and stops flushing if there are enough space. As a result, with a high probability, the number of pages required to be flushed is less than that in a bufferless $IP$.

When an object is removed from its original location, the $IP$ has to update its inverted index table to reflect such change. As described previously, the delete operation requires the $IP$ to scan the entire valid flash storage area and tag the deleted object terms to be invalid. It appears that the delete performance is determined by the size of stored flash data. Our experiment results in Fig. 10 illustrates this trend. The experiment is conducted in the following way: We select a specific object sensor with 10 terms, and perform deletion with different amounts of data loaded in the $IP$, ranging from 0 to 1,600 terms. Initially, the deletion time does not vary much when the number of loaded terms increases. The reason is that the $IP$ has to scan at least one flash page for each index entry, no matter how many terms have already been stored in the flash. When the term number continues to grow, some index entries require more flash page to store the metadata terms. Therefore, the deletion operation has to scan more flash pages. As a result, the time consumption increases accordingly.

Note that deletion does not have to be done each time an object leaves an $IP$'s neighborhood. A simple list can be kept by the $IP$ that records the IDs of the objects that have left. Then, before the $IP$ replies to a query, it removes the objects found in the list from the answer. This way, the user will still have the correct answer. The $IP$ can then perform the deletion in the background when there are no other pending query requests.

## 7.3 Local Query

To evaluate the local query performance, we focus on two main areas: query latency and query accuracy. We first test the performance of the query latency of Snoogle. Then, we demonstrate Snoogle query efficiency via a comparison test that compares the latency performance between Snoogle and a flat structured network. Finally, we evaluate the query accuracy.

### 7.3.1 Query Latency

Query latency is the time taken for a user querying an $IP$ to receive a reply. This includes the time to transmit, process, and reply to a query. To better evaluate our search system, we measure the query latency using common web search characteristics. From [20], the average number of query terms per search is less than 3. We then determine the average time taken to complete a user query comprising of one to four terms. Fig. 11 shows the results. We see that the query response time increases as the number of query terms increase. As mentioned in Section 4.1, multiple flash pages may have to be read from flash memory to determine the

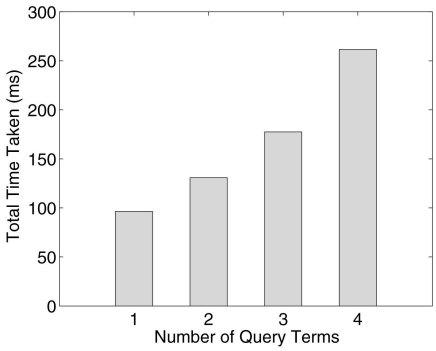



Fig. 11. Time taken for $IP$ to respond to a query.

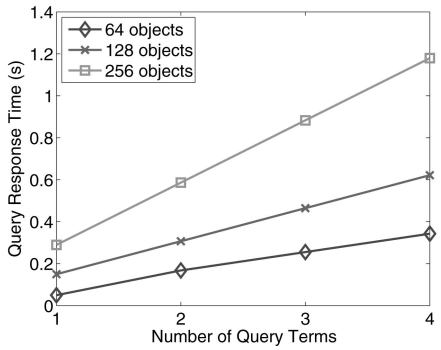



Fig. 12. Time taken for $IP$ to respond to a query.

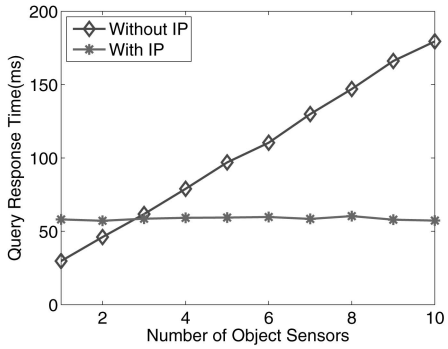



Fig. 13. Query latency with and without $IP$s.

$IDF$ of each query term. This accounts for the increase in query response time.

The above experiment is an example of a local query performance given a typical system setting with 80 objects. To evaluate the scalability performance of the local query, we design the following test: In this test, the $IP$ term index table is configured with 16 entries. As we discussed in Section 2, each object term, after being compressed, is hashed again to get its index in the index table that stores the address pointing to the flash memory. Since we assume an ideal hash function, each term is equally likely hashed to one of 16 indices. Thus, the 16 page chains, pointed by 16 indices, can be considered to have the same length. Based on this observation, we generate a number of synthetic object terms and directly load them into the flash memory of an $IP$. We evaluate the local query performance as the $IP$ starts from lightly loaded to fully loaded.

The experiment results are shown in Fig. 12. We find the local query time is affected by two parameters: 1) the number of query terms; 2) the number of objects. For each query term, the IP needs to scan the flash memory pages that are pointed by a certain entry in the indexing table. Our test shows that it takes around 7 ms to read a page of flash memory to RAM by a TelosB sensor. Multiple terms then require the multiple scans and thus produce the corresponding delay. Note the page scanning delay dominates the total query response time. The similar trend is observed that query time increases when the number of object increases as more terms are stored in flash memory. Overall, the query time is efficient. It takes 1.2 s for an $IP$ to respond a four-term query when there are 256 objects.

#### 7.3.2 Compare to Searching without $IP$s

An alternative searching method is to have users query the objects sequentially and then collect the replied data to find the desired information. This method gets rid of the $IP$. To evaluate, we implement this alternative searching scheme and compared the performance against our Snoogle system. The alternative searching scheme is implemented as follows: A group of sensors are organized to a chained structure. The user always queries the chain head sensor, the queried sensor searches the query term in its memory and puts the results at the preassigned position in the message packet, and then forward the query to the next sensor in the chain. The second object repeats the above searching and puts the results in its preassigned position. This procedure repeats until the last object finishes the query processing. The last sensor directly replies to the user. We believe this is the most efficient way that a general searching scheme can achieve because it requires lowest amount of the message transmission. We select 10 object sensors for both the experiment setups. Each sensor is preloaded with the metadata of one conference paper. The user performs a single term query to both the systems. We measure the user query response time with the number of object sensors changes from 1 to 10. In Fig. 13, we show the difference in query response time in two different searching systems. We see that the query response time in Snoogle system remains relatively constant. The time taken in general searching system, however, increases linearly with the number of objects increases. This proves Snoogle achieves much better scalability than any general searching scheme.

#### 7.3.3 Query Accuracy

Query accuracy in traditional IR uses *precision verses recall* to evaluate the effectiveness of a search system. However, Shah and Croft [21] pointed out that using the mean reciprocal rank (MRR) metric from question answering (QA) was more suitable when performing IR on power constrained or bandwidth limited devices. In QA, the emphasis is to return a single or a very small group of answers in response to a query, and not the return as many relevant answers as possible. In other words, the QA metric places more emphasis on the accuracy of the *ranking* of answers. This is apt for our search system built on sensors. The MRR is defined as

$$MRR = \frac{1}{\text{rank of first correct response}}.$$

For example, let the search system return a ranked answer $(A, C, B)$ where the model answer is $B$. In other words, the correct highest ranked response should be $B$. The MRR for this query is thus $\frac{1}{3} = 0.33$ since the correct answer is three spots off. An answer that matches the model answer will have an MRR of 1.

We first generate three different test files, $q2Term$, $q3Term$, and $q4Term$, each file containing a collection of two, three, and four query terms, respectively. Each collection consists of 20 different questions and model answers. These questions are designed to contain some degree of ambiguity. For example, in our collection, there

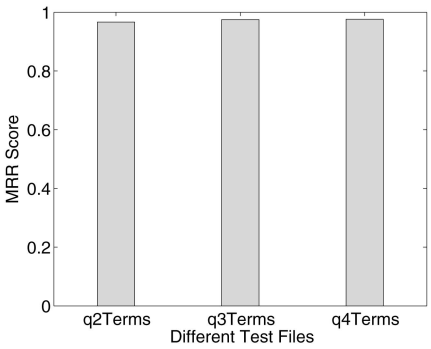



Fig. 14. Accuracy of query answer.

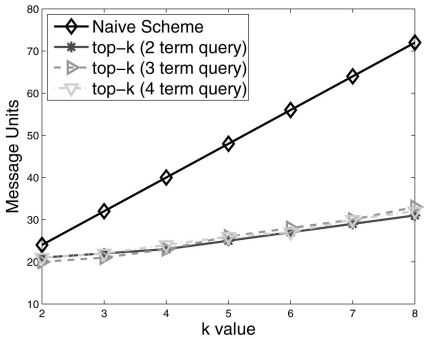



Fig. 15. Message complexity of distributed top-$k$ query.

are two papers with DHT in their title, four papers with Frans Kaashoek listed as an author, and two papers with Jeremy Stribling as an author. Thus, a query "Kaashoek DHT Stribling" will have the model answer the paper titled "Bandwidth-efficient Management of DHT Routing Tables." In other words, the ranked list of answers returned should have this paper as the top-ranked result. Fig. 14 shows the results. We see our search system has a high MRR for different number of query terms. We do not test for only one query term since to derive a model answer, the query term needs to be unique, which is equivalent to a simple grep match.

## 7.4 Distributed Top-$k$ Query

As we discussed in Section 4.3, the message complexity is the major concern in the distributed top-$k$ query. To evaluate the performance of our top-$k$ query scheme, we first perform a simulation based on our own data set to study the message complexity. Then, we estimate the query response time on a larger network setting by using the packet transmission time collected from our experiments. In both the studies, we compare the performance of the proposed top-$k$ query with that of the naive scheme.

### 7.4.1 Message Complexity

In the simulation, we use the same data set which is composed of 80 objects. We evenly and randomly distribute these objects into eight $IP$s (each $IP$ has 10 objects). In this way, we create a testbed for the distributed query with eight $IP$s, which are returned from the $KeyIP$ for the user query (note $m = 8$). In the next step, the user performs the distributed top-$k$ query.

We implement our distributed top-$k$ query scheme on our simulator since our interest is the message complexity only. The rule of determining the message complexity is explained as follows: 1) A single user query to a certain $IP$ is counted as one message unit. 2) The answer with $k$ objects from a certain $IP$ is counted as $k$ message units since the message length grows as $k$ increases. We run the simulations for three different queries with two, three, and four query terms, respectively. We first randomly distribute the objects into eight $IP$s, then run the query and count the message numbers. We repeat this procedure for 100 times for each simulation and calculate the average message count values. For the comparison purpose, we also implement the naive top-$k$ query scheme. Note there is no change in message complexity of naive scheme given variant object distribution and query term numbers.

The simulation results are shown in Fig. 15. As we can see, the performance of naive scheme is significantly worse than that of our distributed top-$k$ query scheme. When $k$ increases by one, the naive scheme needs $m$ more messages (here $m = 8$). Comparatively, the number of extra messages required for our top-$k$ query is much less than $m$. As a result, when $k$ increases to eight, the naive scheme costs 72 messages, while our top-$k$ query only needs 32 messages on average. The figure also shows that the number of query terms has no significant impact on the performance of the distributed top-$k$ query, the performance of two, three, and four term query is very close to each other.

### 7.4.2 Query Response Time

While the above simulation considers performance as measured by the number of message units, an actual application will be more concerned with query response time, especially for a large network. Here, we estimate the distributed query time. Since the $KeyIP$ is a resource-rich computer, we ignore the processing time at the $KeyIP$. The message transmission time thus dominates the query delay. Our experiment shows that the average transmission time $T_p$ for a packet with 68 byte payload is 11.4 ms.

The message complexity of the distributed top-$k$ query is $2(m + k)$, where $m$ is the number of $IP$s having searched terms, and $k$ is the number of top answers the user is looking for. Without loss of generality, we let $k$ be 10 so that the user always wants the top-10 answers. To determine the value of $m$, we consider the following two scenarios: We let $m$ be 20 as the search items are popular, and let $m$ be 5 as the items are nonpopular. We further denote $D$ as the average hop distance between $IP$s and the $KeyIP$. A larger $D$ indicates a larger network.

As we described in Algorithm 1, the $KeyIP$ first queries $m$ $IP$s for their top-ranked answers. After collecting the responses from $m$ $IP$s, the $KeyIP$ sequentially queries up to $k$ $IP$s (that have higher scores than the rest $m - k$) until the top-$k$ answers are found. In the worst case, the message complexity is $2(m + k)$. Combining all components, our estimation of the query time for the distributed top-$k$ query is:

$$T_{query} = m \cdot T_P + 2 \cdot D \cdot T_p + 2 \cdot D \cdot T_p \cdot k. \quad (3)$$

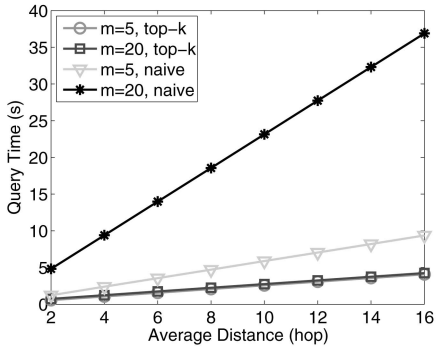



Fig. 16. Query response time of distributed top-$k$ query.

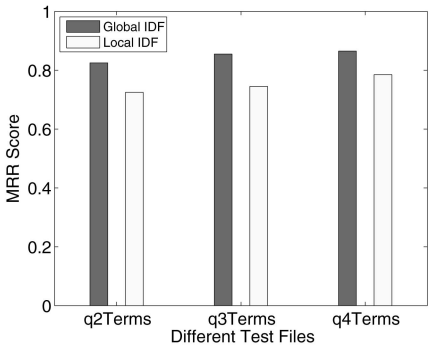



Fig. 17. Query accuracy of distributed top-$k$ query.

The first term is the time to transmit $m$ query messages by the $KeyIP$. Note the $KeyIP$ can continuously send out the messages without waiting for the reply for the specific $IP$. The second term indicates the time duration of the average response time for the query send by the $KeyIP$. The third term is the time taken for the $KeyIP$ to query up to $k$ $IP$s.

For the naive top-$k$ query scheme, each $IP$ has to reply $k$ messages, each of which carries an answer. The total query time can be expressed as:

$$T_{query} = m \cdot T_p + D \cdot T_p + m \cdot k \cdot T_p \cdot D. \quad (4)$$

The first term is the same as in (3). The second term indicates the time taken for the query to reach $IP$s. The third term is the transmission time for the $IP$s to send $k$ messages over $D$ hops.

The estimated query response time for both the schemes is shown in Fig. 16. We find that our distributed top-$k$ query scheme is much more efficient than the naive scheme. When the average hop distance increases from 2 to 16, the query response time of our distributed top-$k$ query grows linearly from 0.7 to 4.2 s. The response time of the naive scheme, on the other hand, grows from 4.8 s to more than 36 s. We also find the value of $m$ has very little affect toward the query response time of our distributed top-$k$ scheme. However, the $m$ value has a large impact on the response time of the naive scheme because every $IP$ out of $m$ members has to respond the query from $KeyIP$ with $k$ answers.

#### 7.4.3 Impact of $IDF$ on Query Accuracy

As we discussed in Section 4.3, we choose to use the global $IDF$ to get accurate merged object rankings. Now, we are interested in the accuracy comparison between the ranking using the global $IDF$ and that using the local $IDF$. We perform the similar simulation as we described in Section 7.3.3. We still use the previous data set with 80 objects that form the testbed with eight $IP$s (each $IP$ has 10 objects). We also use MRR as the metric, and generate three different test files, $q2Term$, $q3Term$, and $q4Term$, each of which contains 20 questions with certain degree of ambiguity.

The test results are illustrated in Fig. 17. We find the MRR scores of the rankings by the global $IDF$ are consistently higher than those of the rankings by the local $IDF$s. The reason can be explained as follows: Once the data set (80 objects) are fixed, the global $IDF$s are determined and will not be affected by the object distribution in $IP$s. Therefore, the merged rankings by using the global $IDF$ are also determined. However, the local $IDF$ values depend on the local document frequency ($DF$), which is affected by the distribution of the objects. Thus, the model answer may not top the merged rankings by using the local $IDF$ if the matching query term has a small local $IDF$ value (due to a large local $DF$) so that the matching term cannot contribute much weight in the overall score.

### 7.5 Security Overhead for User Query

Finally, we add the authentication module to the $IP$ and test the performance of private object query. We use our ECC public key cryptosystem primitive written for TelosB motes. Our extensive optimization allows TelosB mote to efficiently perform ECC public key operation. Our experiment shows it only takes 1.4 s to do a point multiplication. To the best of our knowledge, this is the best ECC performance achieved on TelosB motes by academic implementations. When the user queries the private objects, the user's identity and access privilege have to be verified. The 160-bit ECC based authentication is performed for the verification purpose. The user query response time is presented in Fig. 18. To query a private object, the user waits around 4.9 s to pass the authentication check. Obviously, the authentication time dominates the overall response time. This is because that the ECC based authentication scheme requires three ECC point multiplications, which contribute more than 90 percent of the overall delay.

## 8 System Limitations

**Communication reliability.** From our experience in building the prototype system, we notice that dropped messages are a

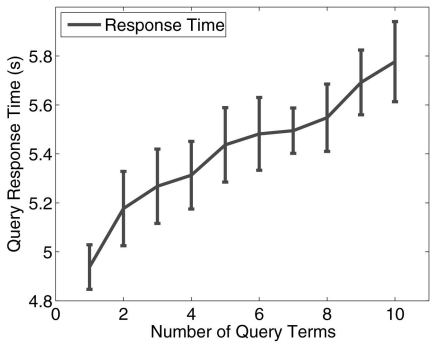



Fig. 18. User perceived private object query response time.

major concern. This occurs when $IP$s report to the $KeyIP$ for record updates, and also during the sending and replying to beacons by an $IP$. This may cause an object sensor to appear missing while moving between $IP$s. In other words, the old $IP$ may have already deleted a leaving object due to the time-out, but the new $IP$ does not detect the object sensor due to packet loss during beacon sending and reply. Therefore, it is desirable to implement a reliable communication mechanism in the application layer or, if possible, in the transport layer, such as acknowledge and retransmission.

**System scalability.** Our Snoogle system design does not limit the number of $IP$s and $KeyIP$s. When the number $IP$s is getting larger (e.g., the network spans several buildings in a district), we can certainly deploy multiple $KeyIP$s, each of which can serve a number of $IP$s in a certain area as we discussed in this paper. Since $KeyIP$s are resource-rich devices, the information sharing and exchanging among $KeyIP$s is beyond the scope of this paper.

Given the limited hardware resources, an individual $IP$ domain has its capacity limit. As we have discussed in the previous section, each object term takes four-byte flash space. Considering 20 terms for each object on average and 1 MB flash memory space, each $IP$ in Snoogle can support more than 10,000 objects in its neighborhood. Given the limited sensor transmission range (normally 100 ft), we believe that the scale of 10,000 objects can support most applications. When the amounts of object are getting larger (within a certain $IP$), however, the query accuracy may be affected for a query with multiple terms. The reason is that the RAM space in $IP$ is very limited in our prototype system and cannot hold all intermediate results while calculating the score of relevancy given a multiterm query. One possible solution is to select a more powerful $IP$ device with a larger RAM size for the object populated areas.

**Mobility support.** While Snoogle supports the search for a mobile object, it does not track a moving object in real time. Due to the power constraints in both $IP$s and object sensors, Snoogle cannot afford very frequent beacon or timer mechanism; thus, an $IP$ may not immediately detect a moving object in its neighborhood. As a result, a snapshot of the system view does not necessarily give accurate moving object locations. However, once the object stops at a certain place for a certain amount of time (e.g., a beacon cycle), the $IP$ at that location will capture the object and update $KeyIP$ with the new indexed items. Obviously, a large number of moving objects will trigger many index updates from $IP$s to the $KeyIP$, which may cause much battery drain and could be a concern of the $IP$ life cycle. We currently assume there are limited moving objects in the system. We leave the problem of $IP$ power management to our future work.

## 9 Related Work

Effective methods for retrieving data have been studied in sensor networks [22], [23]. However, searching in sensor networks has been primarily restricted to numeric data and has not been expanded to handle textual data.

Indoor localization research shares similarities with Snoogle in that sensors are attached to mobile objects [24], [25], [7]. However, most localization research focuses on allowing a sensor to determine its location. One exception is MAX [26] which extends the localization idea to finding objects. In MAX, a user can query a particular object attached with a sensor through an interface and receive hints on where the object can be found, i.e., "top shelf on third room." However, the search functions in MAX are more akin to the `grep` function, determining the presence or absence on a sensor in a particular location. The user, in general, has to know in advance *what* he is looking for, e.g., "my cellphone." Searching in Snoogle is different since a user can discover new knowledge by searching using some general terms and obtain a ranked list of related matches. This is done by adopting information retrieval research into sensor networks. In addition, the security system proposed in MAX does not provide a fine-grained and flexible access control mechanism.

The architecture for our $IP$ follows improvements in low level flash storage. One early work by [27] introduced a file system especially tailored for sensors, providing common file system primitives like append, delete, and rename. While a sensor file system can perform the functionalities of our $IP$, our $IP$ architecture emphasizes good indexing and query response time but not file system functionalities. In this regard, our $IP$ architecture is closer to MicroHash [28] which focuses on efficient indexing of numeric data. Our architecture differs from MicroHash in that we allow indexing of arbitrary kinds of terms, not just numeric ones, and we adopt information retrieval algorithms to reply to queries.

The distributed top-$k$ query discussed in this paper is related to the top-$k$ operations [29], [30] in sensor networks and the distributed top-$k$ query [3] in a general network. Unlike prior work which used extensive simulation for evaluation, we used a combination of actual implementation and simulation to evaluation our top-k algorithm.

Different from Microsearch [31] that allows users to do textual search in the local storage of a stand-alone small device, this paper addresses the searching challenges over a large scale sensor network. The Snoogle system presented in this paper is an extension and enhancement of our previous work [32].

## 10 Conclusion

In this paper, we presented Snoogle, an information retrieval system built on sensor networks. Our system reduces communication costs by employing compressed Bloom filter on object data, while maintaining a low rate of false positives. We also introduced a flexible security method using public key cryptography that protects user privacy. Our current implementation incurs a five second latency. Currently, we are working on different techniques to further reduce the security latency.

## Acknowledgments

The authors would like to thank all the reviewers for their insightful comments and kind guidance to improve the paper. This project was supported in part by US National Science Foundation grants CNS-0721443, CNS-0831904, and CAREER Award CNS-0747108. This manuscript presents the extended work based on our former publication "Snoogle: a search engine for the physical world" in INFOCOM 2008.

## REFERENCES


[1] B. Hull, V. Bychkovsky, K. Chen, M. Goraczko, A. Miu, E. Shih, Y. Zhang, H. Balakrishnan, and S. Madden, "Cartel: A Distributed Mobile Sensor Computing System," *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys '06)*, 2006.

[2] R.K. Ganti, P. Jayachandran, T.F. Abdelzaher, and J.A. Stankovic, "SATIRE: A Software Architecture for Smart Attire," *Proc. Int'l Conf. Mobile Systems, Applications and Services (MobiSys '06)*, 2006.

[3] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *Proc. 20th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, pp. 102-113, 2001.

[4] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel, "The Intel® Mote Platform: A Bluetooth-Based Sensor Network for Industrial Monitoring," *Proc. Fourth Int'l Symp. Information Processing in Sensor Networks (IPSN '05)*, 2005.

[5] Crossbow Technology, Inc., "Wireless Sensor Networks," http://www.xbow.com/Products/Wireless_Sensor_Networks.htm, 2010.

[6] N. Bulusu, J. Heidemann, and D. Estrin, "GPS-Less Low Cost Outdoor Localization for Very Small Devices," *IEEE Personal Comm.*, vol. 7, no. 5, pp. 28-34, Oct. 2000.

[7] N.B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket Location-Support System," *Proc. Sixth Ann. Int'l Conf. Mobile Computing and Networking*, 2000.

[8] N. Bulusu, D. Estrin, L. Girod, and J. Heidemann, "Scalable Coordination for Wireless Sensor Networks: Self-Configuring Localization Systems," *Proc. Sixth IEEE Int'l Symp. Comm. Theory and Application*, July 2001.

[9] A. Savvides, C. Han, and M. Strivastava, "Dynamic Fine-Grained Localization in Ad-Hoc Networks of Sensors," *Proc. Seventh Ann. Int'l Conf. Mobile Computing and Networking*, 2001.

[10] D. Moore, J. Leonard, D. Rus, and S. Teller, "Robust Distributed Network Localization with Noisy Range Measurements," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems*, 2004.

[11] B.H. Bloom, "Space/Time Trade Offs in Hash Coding with Allowable Errors," *Comm. ACM*, vol. 13, no. 7, pp. 422-426, 1970.

[12] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide Area Web Cache Sharing Protocol," *Proc. ACM SIGCOMM*, 1998.

[13] M. Mitzenmacher, "Compressed Bloom Filters," *Proc. 20th Ann. ACM Symp. Principles of Distributed Computing*, 2001.

[14] J. Li, B.T. Loo, J.M. Hellerstein, M.F. Kaashoek, D. Karger, and R. Morris, "On the Feasibility of Peer-to-Peer Web Indexing and Search," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS '03)*,

[15] J. Callan, "Distributed Information Retrieval," *Advances in Information Retrieval*, pp. 127-150, Kluwer Academic Publishers, 2000.

[16] J.C. French, A.L. Powell, J.P. Callan, C.L. Viles, T. Emmitt, K.J. Prey, and Y. Mou, "Comparing the Performance of Database Selection Algorithms," *Proc. Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, 1999.

[17] H. Wang, B. Sheng, C.C. Tan, and Q. Li, "WM-ECC: An Elliptic Curve Cryptography Suite on Sensor Motes," Technical Report WMCS-2007-11, College of William and Mary, 2007.

[18] H. Wang, B. Sheng, C.C. Tan, and Q. Li, "Comparing Symmetric-Key and Public-Key Based Schemes in Sensor Networks: A Case Study for User Access Control," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS)*, June 2008.

[19] Mysaifu, www2s.biglobe.ne.jp/dat/java/project/jvm/, 2010.

[20] B.J. Jansen, A. Spink, J. Bateman, and T. Saracevic, "Real Life Information Retrieval: A Study of User Queries on the Web," *ACM SIGIR Forum*, vol. 32, pp. 5-17, 1998.

[21] C. Shah and W.B. Croft, "Evaluating High Accuracy Retrieval Techniques," *Proc. ACM Special Interest Group on Information Retrieval (SIGIR)*, 2004.

[22] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TinyDB: An Acquisitional Query Processing System for Sensor Networks," *ACM Trans. Database Systems*, vol. 30, pp. 122-173, 2005.

[23] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards Sensor Database Systems," *Proc. Second Int'l Conf. Mobile Data Management (MDM '01)*, pp. 3-14, 2001.

[24] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "The Active Badge Location System," technical report, 1992.

[25] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, "The Anatomy of a Context-Aware Application," *Proc. Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking*, 1999.

[26] K.-K. Yap, V. Srinivasan, and M. Motani, "MAX: Human-Centric Search of the Physical World," *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)*, 2005.

[27] H. Dai, M. Neufeld, and R. Han, "ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes," *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)*, 2004.

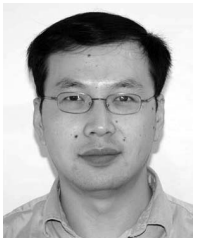
[28] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W.A. Najjar, "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, 2004.

[29] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, V. Tsotras, M. Vlachos, N. Koudas, and D. Srivastava, "The Threshold Join Algorithm for Top-k Queries in Distributed Sensor Networks," *Proc. Second Int'l Workshop Data Management for Sensor Networks (DMSN '05)*, pp. 61-66, 2005.

[30] M. Wu, J. Xu, X. Tang, and W.-C. Lee, "Top-k Monitoring in Wireless Sensor Networks," *IEEE Trans. Knowledge and Data Eng.*, vol. 19, no. 7, pp. 962-976, July 2007.

[31] C.C. Tan, B. Sheng, H. Wang, and Q. Li, "Microsearch: When Search Engines Meet Small Devices," *Proc. Sixth Int'l Conf. Pervasive Computing*, May 2008.

[32] H. Wang, C.C. Tan, and Q. Li, "Snoogle: A Search Engine for the Physical World," *Proc. IEEE INFOCOM*, Apr. 2008.

**Haodong Wang** received the PhD degree in computer science from the College of William and Mary. He is an assistant professor in the Department of Mathematics and Computer Science at Virginia State University. His research interests include information assurance in cyber-physical systems, privacy preserving and user access control in sensor networks, efficient information storage, search and retrieval in pervasive computing, and MAC design in IEEE802.11 wireless LAN.

**Chiu C. Tan** received the BS degree in computer science and the BA degree in economics (honors) from the University of Texas at Austin in 2004. He is currently a graduate research assistant at the Department of Computer Science, College of William and Mary. His research interests include ubiquitous computing, embedded systems, large scale RFID systems, vehicular networks, and wireless security.

**Qun Li** received the PhD degree in computer science from Dartmouth College. He is an assistant professor in the Department of Computer Science at the College of William and Mary. His research interests include wireless networks, sensor networks, RFID, and pervasive computing systems. He received the US National Science Foundation (NSF) Career award in 2008.