Snoogle: A Search Engine for the Physical World

Haodong Wang, Chiu C. Tan, and Qun Li
Department of Computer Science
College of William and Mary
Email:{wanghd,cct,liqun}@cs.wm.edu

***Abstract*—Hardware advances will allow us to embed small devices into everyday objects such as toasters and coffee mugs, thus naturally form a wireless object network that connects the object with each other. This paper presents Snoogle, a search engine for such a network. Snoogle uses information retrieval techniques to index information and process user queries, and compression schemes such as Bloom filters to reduce communication overhead. Snoogle also considers security and privacy protections for sensitive data. We have implemented the system prototype on off-the-shelf sensor motes, and conducted extensive experiments to evaluate the system performance.**

I. Introduction

A pervasive computing environment entails embedding small computing devices into everyday objects such as tables and shelves. These small devices allow us to store and retrieve information regarding the underlying physical object. For instance, consider a collection of document binders. Each binder is embedded with a device containing a short description of the contents of that binder. This description can be created through input devices such as a digital pen which can translate a person's handwriting onto text, or by a miniature RFID reader that scans every RFID enabled document place into the folder. A user wanting to find a particular document can query each binder's embedded device to learn of the contents, and then retrieve the appropriate binder.

In this paper we present Snoogle, a search engine that allows users to efficiently search for information in a pervasive computing environment. We assume that these small devices are already embedded in everyday objects, and each device has limited processing, storage and communication ability. We use the terms "sensor" or "mote" to describe such a resource limited device. We also assume that an effective data input mechanism is used to store the necessary data into the device.

Snoogle adheres to the design principal that *information pertaining to an object on the object itself*. In other words, information describing a binder should be stored on the binder itself, and not on a remote server. This improves robustness since a wireless connection to a remote server may not be available but information stored on the binder can be always be accessed via short range wireless protocols like Blue-tooth. However, adhering to this principal requires novel data storage and management techniques to be implemented on the mote. Furthermore, this principal also require security and privacy protections be tailored for the resource constrained motes.

We make the following contributions in this paper. (1) We designed a complete search system using information retrieval (IR) techniques on a resource constrained platform. We are unaware of any previous efforts in designing IR systems on sensor hardware. (2) We investigated the use of bloom filters and compressed bloom filters to reduce overall communication costs. (3) A distributed top-k query algorithm is presented for efficient querying. (4) Our system design provides distributed security and privacy protections, and this module is implemented in our system prototype for evaluation. (5) A combination of a working prototype and simulation is used for evaluation.

The rest of the paper is presented as follows. Sections II and III describe the Snoogle system architecture and query resolution algorithms respectively. Section IV examines how Snoogle provides mobility, security and privacy support. Section V contains the evaluation of our system, Section VI discusses some limitations, and Section VII presents the related work. Finally, Section VIII concludes.

II. System Design

A. System Components

Snoogle consists of three components: object sensors, Index Points (IPs) and Key Index Points ($KeyIP$s). An object sensor is a mote attached to a physical object, and contains a textual description of the physical object. This description is determined by the object owner. The object sensor can be either static or mobile, depending on whether the corresponding physical object is stationary or mobile.

An IP is a static sensor that is associated with a physical location, for example, a particular room in an office building. IPs are responsible for collecting and maintaining the data from the object sensors in their vicinity. The IP hardware is similar to an object sensor, but with larger storage capacity. A collection of IPs forms a homogenous mesh network.

The $KeyIP$ collects data from different IPs 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.

B. System Architecture

Snoogle adopts a two-tier hierarchical architecture depicted in Fig. 1. The lower tier involves object sensors and IPs. Each IP manages a certain area within its transmission range. Object sensors register themselves and transmit the object description metadata to the specific IP. IPs are responsible for building the inverted indexes for local search.

On the upper tier, IPs have their dual roles. First, IPs forward the aggregated object information to the $KeyIP$ so

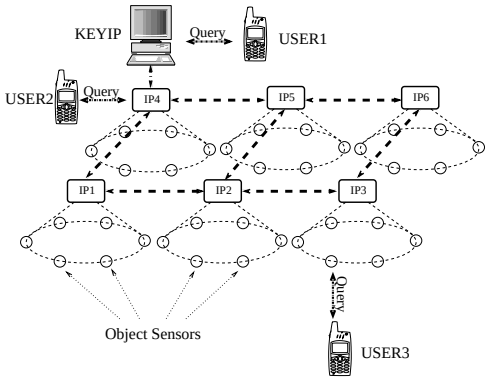

Fig. 1. Overview of Sensors, IP and $KeyIP$ Architecture

that the $KeyIP$ can return a list of IPs that are most relevant to a certain user query. Second, IPs also provide the message routing for the traffic between IPs, $KeyIP$, and users. The $KeyIP$, considered as the sink of the IP mesh network, holds the global object aggregation information reported by each IP. While Snoogle does not restrict the number of $KeyIP$s, for the simplicity, this paper only considers the single $KeyIP$ setup.

Users query Snoogle using a portable device such as a cell phone. 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 search for objects at a specific location. A user performs a distributed query when he queries the $KeyIP$. The distributed query capability allows Snoogle to scale since users do not need to flood every IP to find a particular object. We discuss querying in further detail in the next section.

C. Data Processing in Object Sensors

Each object sensor contains two types of data, *payload data* and *metadata*. *Payload* is a short description of the attached physical object. *Metadata* is a representation of the *payload*. For example, the *payload* of an object sensor attached to a folder can be a short note describing the contents of the folder. The *metadata* is a set of tuples, $\{term_1 : freq_1 : id_1\} \cdots \{term_n : freq_n : id_n\}$, where term is a single word describing the *payload*, and $freq$ indicates the importance of this term in describing the *payload*. A user storing information into an object sensor is responsible for sending the *payload* and *metadata*. To minimize the data transmission cost, the data in the object sensor can also be pre-compressed using compression schemes described in the next section.

D. Data Processing and Storage at IPs

IPs in Snoogle have two data processing roles. First, IPs collect data from object sensors within their range and organize the data into an inverted index. Due to reliability and space concerns, the inverted index table in the IP is stored in the on-board flash memory of sensors rather than RAM. Second, IPs periodically send aggregated update information that reflects the object dynamics in its area, to the $KeyIP$ so that the $KeyIP$ can maintain a consistent inverted index of IPs. IPs performs the following three data operations.

Insert: This operation is executed when a new object comes into the IP's region and sends the metadata to the IP. The IP stores the new metadata 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 sensor is no longer associated with the previous IP. The IP then performs a "delete" operation to remove all the metadata of the leaving object from its inverted table.

Modify: This operation is performed when there is a change in the object sensor's data. When this happens, the object sensor sends a modification request to the IP. Since the IP inverted table is 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 improve the flash storage efficiency, basing on the observation that the IP only stores the metadata of the objects, instead of the whole payload data which has to be considered in the general storage media. We take advantage of the small granularity write capability of the NOR flash (TelosB on-board flash memory) and allow IPs to be able to append the object metadata sequentially in the flash memory without extra *read and write* required in the NAND flash. In this way, the flash erasing operation (once the flash is full) is kept to the minimum. In addition, we also implement a "delete" function that efficiently invalidates the metadata associated with an object sensor. We perform the "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 sensor sends its id and metadata to the IP, the information is first stored in a buffer in RAM. 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. Fig. 2 illustrates the IP storage architecture.

When the 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

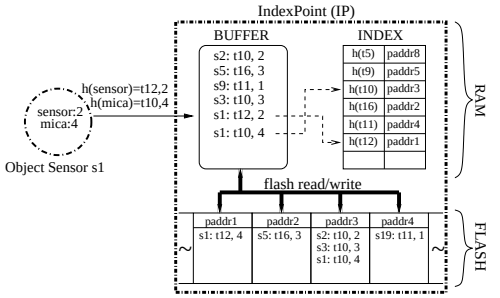

Fig. 2. Sensor S1 sending data to IP

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 sensor ids is obtained, then a ranked list of sensor ids that best match the query is derived using an information retrieval (IR) algorithm further elaborated in the next section.

Finally each IP will periodically send the updated metadata terms and sensors, which reflect the object dynamics in the region, to the $KeyIP$. The $KeyIP$ stores the data and checks for inconsistency. This inconsistency arises when sensors moved from one IP (IP_1) to another (IP_2) before IP_1 have a change to update its record. Since all sensors have a unique id, this inconsistency is easily detected by the $KeyIP$. The $KeyIP$ then informs both IPs verify the sensor data. For example, both IP_1 and IP_2 report having sensor 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 delete s_1 from its inverted index table.

E. Communication Compression

A Bloom filter [1] is used in Snoogle to compress groups of terms together. A Bloom filter with a m-bit array and k independent hash functions is 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. Any "0" indicates there is no match. However, a 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 collision, we use 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

$$(1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-kn/n})^k. \quad (1)$$

When m and n are fixed, the optimal false positive rate can be achievable when [2]

$$k = ln2 \cdot \frac{m}{n} \quad (2)$$

Bloom filters can be further compressed to achieve better transmission efficiency [3]. This is based on the observation that a 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 the compression only works ($z < m$) when there are less "1"s than "0"s (or in reversed case). Mitzenmacher [3] 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 tells us an optimal Bloom filter almost cannot be compressed. It also means there is trade-off between false positive and compression ratio. To gain transmission efficiency, we have to sacrifice the false positive rate. Mitzenmacher [3] continued to point out that the procedure of compressing a Bloom filter is actually equivalent to hash 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. Suppose 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 [4] 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. Actually, with carefully chosen parameters, we can lower the false positive rate to an acceptable level. As we will describe in the evaluation section, given the data set with 1512 words, and compressed Bloom filter size of 16 bits, the false positive rate is only about 2.3%.

III. Performing Query

As mentioned earlier, 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 IPs 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 if the user has an approximate idea where the object might be found. For example, a user wants to find a particular magazine, but only if it is within a short distance from where he is currently at. Thus, he only queries the IP near him by sending a few terms that describe this magazine. The IP evaluates the answers to the user. Each answer is the id of an object that best matches the user query. The user can then query the sensor directly, or physically find the sensor and hence 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 IPs that contain objects that best

match the query, where m is a system parameter. The user then perform the distributed top-k query from the returned m IPs and find the satisfied answers.

A. Improving Query Accuracy

When a user queries an IP, he receives a ranked list of sensor ids that best match his query from the IP as his answer. This ranking is derived from a score for each sensor 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 sensors, (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 sensor is determined using the TF/IDF weighing algorithm from IR research. The intuition behind TF/IDF is that the importance of a term in describing a sensor is based on two considerations. The first is the number of times that term appears in that sensor, the TF. The more often a term appears in the sensor, the more relevant that term is in describing that sensor. In our system, the TF value is given as part of the metadata of the sensor.

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

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

The idea here is that if a term appears in many sensors found within an IP's neighborhood, it is less important. Consider the extreme case where a term appears in every sensor under an IP. Then, any one of the sensors returned will contain that term, making that term not descriptive of any one sensor at all. To get the IDF value, we need the total number of sensors and the number of sensors containing the term. The first one is easy to get since an IP knows all sensors in its neighborhood. The second value is acquired while processing the query at an IP. Given 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 IP 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 the IP in the distributed query. We initially considered CORI weighing algorithm [5] when a user queries the $KeyIP$, but there was no noticeable improvement. Thus we use a simple TF/IDF algorithm throughout this paper.

B. 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 top-k query scheme is for the user to perform a top-k query for each of the m IPs 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.

Algorithm 1 Distributed Top-k Query Algorithm

1: Input: k IPs: $IP_1, IP_2, \cdots, IP_m$
2: Output: top-k answers: $Obj_1, Obj_2, \cdots, Obj_k$
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 $Obj_1, Obj_2, \cdots, Obj_k$

Our distributed top-k query algorithm is shown in Algorithm. 1. The basic idea can be explained as follows. Upon receipt of the list of m ranked IPs, the user queries each IP for the most relevant object, denoted as $ta_i, 1 \leq i \leq m$. The user 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 top weighing objects from all m IPs, the user does a sorting in the descending order of the object weight, and obtains a new array that $a[1].weight \geq a[2].weight \geq \cdots \geq a[m].weight$. By now, the first top-$k$ answer: $a[1].obj$, is immediately available. The next phase is to search for the remaining answers. The user 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 IPs, it is possible for IP $a[1].ip$ to solely hold objects with weights no less than $a[2].weight$, so there is no reason to firstly query other IPs. Ignoring all the committed objects (i.e., they are definitely top-k objects), each IP has a new top 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.

To bound the number of messages transmitted in the process, we make the following observations. First, each IP in total sends out at most one object that will not appear in the top-k list. Therefore, the number of messages sent by all the IPs is at most $m + k$ including the top-k objects and those

"useless" 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 IPs is bounded by the number of received objects, which is at most $m + k$. Combining the two observations, we conclude that the number of messages in this process is at most $2(m + k)$. Compared to the message complexity of $m \cdot k$ in naive scheme, obviously our distributed top-k query scheme is much more efficient.

IV. Mobility and Security

A. Supporting Mobile Objects

As objects can be mobile, there will inevitably be objects moving in and out of an IP's neighborhood. Snoogle uses a combination of *beacon* and *timer* methods to ensure an IP maintains up-to-date information

In the *beacon* method, the IP will periodically broadcast a beacon that identifies itself. An object sensor in the neighborhood that receive this beacon will compare it against the previous beacon. Matching beacons indicates that the object is still covered by the same IP, and the sensor does nothing. Otherwise, the sensor will report its 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 is gone, and then deletes the all data of the object sensor from its storage. The *beacon* and *timer* methods can be regarded as a "$pull$" or a "$push$". In the *beacon* method, IPs $pull$ the status information from the object sensors. In the timer method, object sensors $push$ their status to IPs.

The *beacon* scheme consumes less energy than the *timer* method since 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.

B. Providing Security and Privacy

Since Snoogle is built on sensors, it shares all common security threats with other applications in sensor networks. Besides, Snoogle also poses unique security and privacy requirements in searching. The concern is that a search engine like Snoogle may violate personal privacy by revealing object information to others. For example, a user may not want his private object (e.g., a DVD movie) to be searchable by strangers, but only his friends and himself. In another example, a thief can query Snoogle for a list of locations most likely to have valuables like laptops. He can then optimize his haul by targeting the highest ranked location first.

To address these concerns, Snoogle must have a security mechanism to prevent the private objects from being searched by unauthorized users. In other words, a user needs to be authenticated before he can search private objects. We adopt the public key cryptography rather than the symmetric key scheme to have a clean user interface and a simple key management. Recent research [6], [7], [8] have demonstrated that public key schemes are feasible for sensor nodes. We developed an Elliptic Curve Cryptography (ECC) public key scheme for Snoogle. The reason we choose ECC over more popular RSA is that ECC can be more efficiently implemented in resource constrained sensors. Due to the space limit, we omit the discussion of ECC implementation on TelosB motes. The interested reader is referred to our technical report [9] for more details. On TelosB sensor motes, it takes 1.4s to generate a public key. In Snoogle, the access control is performed at the IP instead of at $KeyIP$ in a distributed fashion.

We provide security protections in Snoogle by adding a security tag to the private object. 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 a Certification Authority like Snoogle administrator. The IP first verifies the user certificate and the makes sure the corresponding *OwnerID* and *GroupMask* matching with the object tag. Then, the IP uses the derived user public key (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 IP and the user the key to establish a secure channel. This key can also be used to achieve the user privacy. The user can simply encrypt his query terms by using the key so that no one can learn the query content.

V. Performance Evaluation

We implement a prototype of Snoogle, including object sensors, IPs and user module, on TelosB motes, a research platform developed by Berkeley. TelosB hardware features a lower-power TI MSP430 16-bit micro-controller with 10KB RAM and 48KB ROM. The on-board IEEE 802.15.4/ZigBee compliant radio transceiver facilitates the wireless communication with other IEEE 802.15.4 compliant devices. TelosB also has an on-board flash memory with 1MB 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 522MHz ARM920T PXA270 processor, 64MB RAM and 128MB flash memory. The software of IPs, object sensors and user module are written by NesC language on TinyOS version 1.1.15.

To better discern the performance of the system, we break the search system down into individual components and evaluate each separately. We mainly focus on object sensor and IP interaction. The reason is twofold. First, both the sensor and IP are power constrained and computationally challenged devices, while the $KeyIP$ can be a resource-rich device. This makes the performance of the object sensor and IP crucial for the validity of the system. Second, we believe most user queries will be directed towards IPs, rather towards the $KeyIP$. This is mainly due to privacy restrictions. For instance, a user looking for his keys will most likely start querying familiar locations rather than the entire building.

We derive our workload by collecting information from various conference abstracts. The title, authors and affiliations of each accepted entry becomes the metadata terms in each sensor. We use the IR definition of TF to obtain the weight of each metadata term. This yields a workload sufficient for about 80 sensors, each of which has about 15 to 25 unique words on average.

A. Data Input and Maintenance at IPs

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 sensors within its range, and obtain their metadata. Fortunately, this initialization phase occurs rarely since our 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

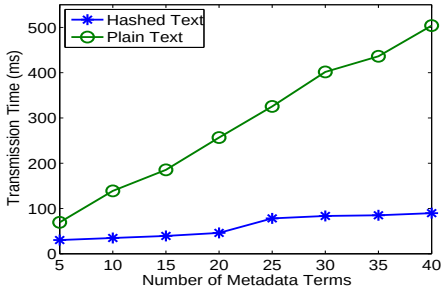

Fig. 3. Time taken to transmit metadata to IP

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. 3 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 $90ms$ to collect 40 compressed terms. However, it requires more than 5 times of amount of time to transfer 40 uncompressed terms.

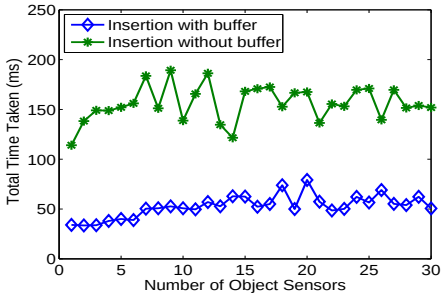

Fig. 4. Insertion performance with buffer and without buffer at IP

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. Flash memory, unlike conventional disk, is written on a per page basis, usually on the magnitude of 256-512 bytes per page. When there are multiple sensors wanting to send data to an IP, the IP will have to periodically halt transmission to flush the coming data into flash. This lengthens the time taken for an object sensor to successfully transmit data to an IP, especially during the initial stage when a group of object sensors upload the data to the IP. To solve this issue, the IP maintains a small buffer in RAM , e.g., 256 bytes, to buffer sensor data before flushing to flash. The IP therefore 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.

We set the buffer size with 256 bytes, equivalent to the page size of the 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 the both 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. 4. It clearly shows that each object sensor waits significantly less amount of time when the IP uses the buffer.

We also 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,

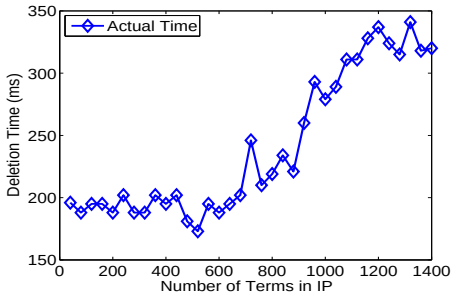

Fig. 5. The amount of time to delete an object with 10 terms.

and stops flushing if there are enough space. As the 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 IP can do a "delete" operation to remove a certain object from its storage. The "delete" operation requires the IP to scan the entire valid flash storage area and tag the deleted object terms to be invalid. It is not difficult to suggest that "delete" performance is determined by the size of stored flash data. Our experiment results, as shown in Fig. 5, exactly show 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 1600 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 the result, the time consumption increases accordingly.

Note that deletion does not have to be done each time a sensor leaves an IP's neighborhood. A simple list can be kept by the IP that records the ids of sensors that have left. Then, before the IP replies to a query, it removes the sensors 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.

B. Local Query

To evaluate the local query performance, we focus query latency. We first test the performance of the query latency of Snoogle. Then, we demonstrate the Snoogle query efficiency by a comparison test that compares the latency performance between Snoogle and a flat structured network.

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 [10], 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. 6 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 IDF of each query term. This accounts for the increase in query response time.

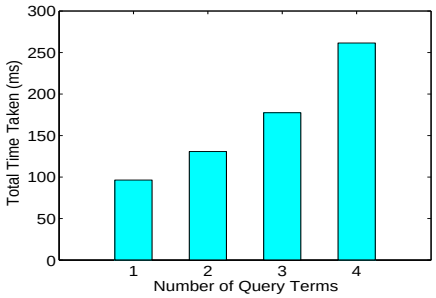

Fig. 6. Time taken for IP to respond to a query

2) Compare to searching without IPs: An alternative searching method is to have users query the sensors 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 pre-assigned position in the message packet, and then forward the query to the next sensor in the chain. The 2^{nd} sensor repeats the above searching and puts the results in its pre-assigned position. This procedure repeats until the last sensor 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 sensors for the both experiment setups. Each sensor is pre-loaded with the metadata of one conference paper. The user performs a single term query to the both systems. We measure the user query response time with the number of object sensors changes from 1 to 10. In Fig. 7, we show the

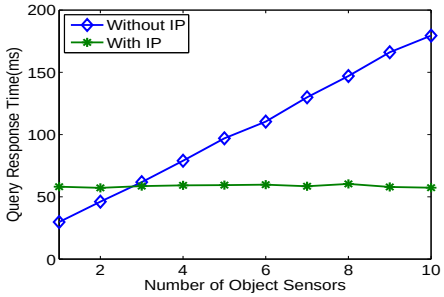

Fig. 7. Query latency with and without IPs

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 then any general searching scheme.

C. Distributed Top-k Query

As we discussed in Section III-B, the message complexity is the major concern in the distributed top-k query. To evaluate the performance of our top-k query scheme, we use the same dataset, which is composed of 80 objects. We evenly and randomly distribute these objects into eight IPs (each IP has 10 objects). In this way, we create a testbed for the distributed query with eight IPs, 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.

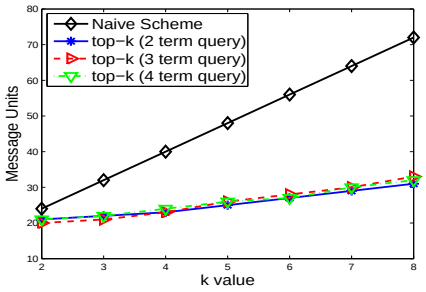

Fig. 8. Message complexity of distributed top-k query.

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 IPs, 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. 8. 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 the 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.

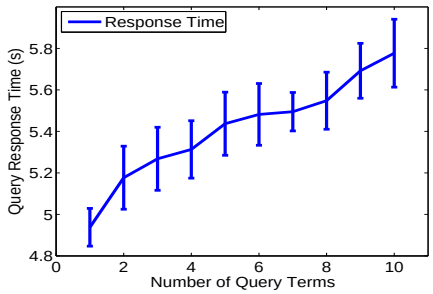

Fig. 9. User perceived private object query response time.

D. Security Overhead for User Query

Finally, we add the authentication module to the IP and test the performance of private object query. We used an ECC public key cryptosystem designed for TelosB motes. Our extensive optimization allows TelosB mote to efficiently perform ECC public key operation. Our experiment shows it only takes 1.4s 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. 9. To query a private object, the user waits around 4.9s to pass the authentication check. Obviously, the authentication time dominates the overall response time. This is because that the ECC based authentication scheme requires 3 ECC point multiplications, which contribute more than 90% of the overall delay.

VI. System Limitations

Communication Reliability In the course of running our experiments, we observed that dropped messages has larger effect on performance than originally expected. Dropped messages resulted in occasional objects that suddenly disappear and reappear at a different location. This occurs when an IP has deleted a leaving object, but the new IP does not detect the moving object due to packet loss during beacon sending and reply. This suggest that a reliable communication mechanism might be useful.

System Scalability. Our Snoogle design utilizes one $KeyIP$ to manage all the IPs. In practice, multiple $KeyIP$s can be deployed for scalability. For example, in an office complex consisting of several buildings, each building can have its own $KeyIP$. Since $KeyIP$s are resource rich devices, less constraints are placed on techniques for information exchange.

Another concern for scalability is that a single IP is insufficient when there are too many objects. An IP in Snoogle uses 4 bytes of flash memory to store each descriptive term. Assuming that an object can be described with 20 terms, an IP with 1MB flash can support over 10000 objects, a relatively large number. For applications which involve extremely large number of objects, a more powerful IP can be used.

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 IPs and object sensors, Snoogle cannot afford very frequent beacon or timer mechanism so that an IP may not immediately detect a moving object in its neighborhood. Therefore, 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 IPs 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 and reserve the IP power management in our future work.

VII. Related Work

Effective methods for retrieving data has been studied in sensor networks [11], [12]. However, searching in sensor networks are primarily restricted to numeric data, and have not been expanded to handle textual data.

Indoor localization research shares similarities with Snoogle in that sensors are attached to mobile objects [13], [14], [15]. However, most localization research is focused on allowing a sensor to determine its location. One exception is MAX [16] which extends the localization idea to finding objects. In MAX, a user can query for 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 is 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 network. In addition, the security system proposed in MAX does not provide a fine-grained and flexible access control.

The architecture for our IP follows improvements in low level flash storage. One early work by [17] 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 and not file system functionalities. In this regard, our IP architecture is closer to MicroHash [18] 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. Recent work by [19] can also be considered for an IP.

VIII. 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 sensor data, while maintaining low rates of false positive. 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 latency for security.

Acknowledgments

The authors would like to thank all the reviewers for their helpful comments. This project was supported in part by US National Science Foundation award CCF-0514985 and CNS-0721443.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," in *Communications of the ACM, 13(7):422-426*, 1970.

[2] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol." in *SIGCOMM 1998*.

[3] M. Mitzenmacher, "Compressed bloom filters," in *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing*, 2001.

[4] 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," in *IPTPS03*.

[5] 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," in *Research and Development in Information Retrieval*.

[6] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," in *CHES*, 2004.

[7] A. Liu and P. Ning, "TinyECC: Elliptic curve cryptography for sensor networks," 2005.

[8] H. Wang and Q. Li, "Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper)," in *International Conference on Information and Communication Security (ICICS), LNCS 4307*, Raleigh, NC, Dec. 2006, pp. 519–528.

[9] H. Wang, B. Sheng, C. C. Tan, and Q. Li, "WM-ECC: an Elliptic Curve Cryptography Suite on Sensor Motes," College of William and Mary, Computer Science, Williamsburg, VA, Tech. Rep. WM-CS-2007-11, 2007.

[10] B. J. Jansen, A. Spink, J. Bateman, and T. Saracevic, "Real life information retrieval: a study of user queries on the web," *SIGIR Forum 1998*.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, 2005.

[12] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *MDM 2001: Proceedings of the Second International Conference on Mobile Data Management*. London, UK: Springer-Verlag, 2001, pp. 3–14.

[13] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "The active badge location system," Tech. Rep., 1992.

[14] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, "The anatomy of a context-aware application," in *Mobile Computing and Networking*.

[15] N. Priyantha, A. Chakraborty, and H. Balakrishnan, "The cricket location-support system," in *MobiCom 2000*.

[16] K.-K. Yap, V. Srinivasan, and M. Motani, "MAX: human-centric search of the physical world," in *Sensys 2005*.

[17] H. Dai, M. Neufeld, and R. Han, "ELF: an efficient log-structured flash file system for micro sensor nodes," in *SenSys 2004*.

[18] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar, "MicroHash: An efficient index structure for flash-based sensor devices." in *FAST 05*.

[19] C. C. Tan, B. Sheng, H. Wang, and Q. Li, "Microsearch: To search a world in a grain of sand," in *the Sixth International Conference on Pervasive Computing*, 2008.