Multi-dimensional Range Queries in Sensor Networks[*]

Xin Li[†] Young Jin Kim[†] Ramesh Govindan[†] Wei Hong[‡]

ABSTRACT

In many sensor networks, data or events are named by attributes. Many of these attributes have scalar values, so one natural way to query events of interest is to use a *multi-dimensional* range query. An example is: "List all events whose temperature lies between 50° and 60°, and whose light levels lie between 10 and 15." Such queries are useful for correlating events occurring within the network.

In this paper, we describe the design of a distributed index that scalably supports multi-dimensional range queries. Our *distributed index for multi-dimensional data* (or DIM) uses a novel geographic embedding of a classical index data structure, and is built upon the GPSR geographic routing algorithm. Our analysis reveals that, under reasonable assumptions about query distributions, DIMs scale quite well with network size (both insertion and query costs scale as $O(\sqrt{N})$). In detailed simulations, we show that in practice, the insertion and query costs of other alternatives are sometimes an order of magnitude more than the costs of DIMs, even for moderately sized network. Finally, experiments on a small scale testbed validate the feasibility of DIMs.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems; C.3 [**Special-Purpose and Application-Based Systems**]: Embedded Systems

General Terms

Embedded Systems, Sensor Networks, Storage

*This work is supported in part by NSF grant CCR-0121778 for the Center for Embedded Systems and by a grant from the Intel Corporation

†Computer Science Department, University of Southern California, {xinli, youngjki, ramesh}@usc.edu

‡Intel Research Lab at Berkeley, wei.hong@intel.com

Keywords

DIM, Multi-dimensional Range Queries

1. INTRODUCTION

In wireless sensor networks, data or events will be named by attributes [15] or represented as virtual relations in a distributed database [18, 3]. Many of these attributes will have scalar values: *e.g.*, temperature and light levels, soil moisture conditions, *etc.* In these systems, we argue, one natural way to query for events of interest will be to use *multi-dimensional* range queries on these attributes. For example, scientists analyzing the growth of marine micro-organisms might be interested in events that occurred within certain temperature and light conditions: "List all events that have temperatures between 50°F and 60°F, and light levels between 10 and 20".

Such range queries can be used in two distinct ways. They can help users efficiently drill-down their search for events of interest. The query described above illustrates this, where the scientist is presumably interested in discovering, and perhaps mapping the combined effect of temperature and light on the growth of marine micro-organisms. More importantly, they can be used by *application software* running within a sensor network for *correlating* events and triggering actions. For example, if in a habitat monitoring application, a bird alighting on its nest is indicated by a certain range of thermopile sensor readings, and a certain range of microphone readings, a multi-dimensional range query on those attributes enables higher confidence detection of the arrival of a flock of birds, and can trigger a system of cameras.

In traditional database systems, such range queries are supported using pre-computed indices. Indices trade-off some initial pre-computation cost to achieve a significantly more efficient querying capability. For sensor networks, we assert that a centralized index for multi-dimensional range queries may not be feasible for energy-efficiency reasons (as well as the fact that the access bandwidth to this central index will be limited, particularly for queries emanating from within the network). Rather, we believe, there will be situations when it is more appropriate to build an in-network *distributed data structure* for efficiently answering multi-dimensional range queries.

In this paper, we present just such a data structure, that we call a DIM[1]. DIMs are inspired by classical database indices, and are essentially embeddings of such indices within the sensor network. DIMs leverage two key ideas: *in-network*

[1]Distributed Index for Multi-dimensional data.

data centric storage, and a novel locality-preserving geographic hash (Section 3). DIMs trace their lineage to data-centric storage systems [23]. The underlying mechanism in these systems allows nodes to consistently hash an event to some location within the network, which allows efficient retrieval of events. Building upon this, DIMs use a technique whereby events whose attribute values are "close" are likely to be stored at the same or nearby nodes. DIMs then use an underlying geographic routing algorithm (GPSR [16]) to route events and queries to their corresponding nodes in an entirely distributed fashion.

We discuss the design of a DIM, presenting algorithms for event insertion and querying, for maintaining a DIM in the event of node failure, and for making DIMs robust to data or packet loss (Section 3). We then extensively evaluate DIMs using analysis (Section 4), simulation (Section 5), and actual implementation (Section 6). Our analysis reveals that, under reasonable assumptions about query distributions, DIMs scale quite well with network size (both insertion and query costs scale as $O(\sqrt{N})$). In detailed simulations, we show that in practice, the event insertion and querying costs of other alternatives are sometimes an order of magnitude the costs of DIMs, even for moderately sized network. Experiments on a small scale testbed validate the feasibility of DIMs (Section 6). Much work remains, including efficient support for skewed data distributions, existential queries, and node heterogeneity.

We believe that DIMs will be an essential, but perhaps not necessarily the only, distributed data structure supporting efficient queries in sensor networks. DIMs will be part of a suite of such systems that enable feature extraction [7], simple range querying [10], exact-match queries [23], or continuous queries [15, 18]. All such systems will likely be integrated to a sensor network database system such as TinyDB [17]. Application designers could then choose the appropriate method of information access. For instance, a fire tracking application would use DIM to *detect* the hotspots, and would then use mechanisms that enable continuous queries [15, 18] to track the spatio-temporal progress of the hotspots. Finally, we note that DIMs are applicable not just to sensor networks, but to other deeply distributed systems (embedded networks for home and factory automation) as well.

2. RELATED WORK

The basic problem that this paper addresses — multi-dimensional range queries — is typically solved in database systems using *indexing* techniques. The database community has focused mostly on centralized indices, but distributed indexing has received some attention in the literature.

Indexing techniques essentially trade-off some data insertion cost to enable efficient querying. Indexing has, for long, been a classical research problem in the database community [5, 2]. Our work draws its inspiration from the class of multi-key constant branching index structures, exemplified by k-d trees [2], where k represents the dimensionality of the data space. Our approach essentially represents a geographic embedding of such structures in a sensor field. There is one important difference. The classical indexing structures are *data-dependent* (as are some indexing schemes that use locality preserving hashes, and developed in the theory literature [14, 8, 13]). The index structure is decided not only by the data, but also by the order in which data is inserted. Our current design is not data dependent. Finally, tangentially related to our work is the class of spatial indexing systems [21, 6, 11].

While there has been some work on distributed indexing, the problem has not been extensively explored. There exist distributed indices of a restricted kind—those that allow exact match or partial prefix match queries. Examples of such systems, of course, are the Internet Domain Name System, and the class of distributed hash table (DHT) systems exemplified by Freenet[4], Chord[24], and CAN[19]. Our work is superficially similar to CAN in that both construct a zone-based overlay atop of the underlying physical network. The underlying details make the two systems very different: CAN's overlay is purely logical while our overlay is consistent with the underlying physical topology. More recent work in the Internet context has addressed support for range queries in DHT systems [1, 12], but it is unclear if these directly translate to the sensor network context.

Several research efforts have expressed the vision of a database interface to sensor networks [9, 3, 18], and there are examples of systems that contribute to this vision [18, 3, 17]. Our work is similar in spirit to this body of literature. In fact, DIMs could become an important component of a sensor network database system such as TinyDB [17]. Our work departs from prior work in this area in two significant respects. Unlike these approaches, in our work the data generated at a node are hashed (in general) to different locations. This hashing is the key to scaling multi-dimensional range searches. In all the other systems described above, queries are flooded throughout the network, and can dominate the total cost of the system. Our work avoids query flooding by an appropriate choice of hashing. Madden *et al.* [17] also describe a distributed index, called Semantic Routing Trees (SRT). This index is used to direct queries to nodes that have detected relevant data. Our work differs from SRT in three key aspects. First, SRT is built on single attributes while DIM supports mulitple attributes. Second, SRT constructs a routing tree based on historical sensor readings, and therefore works well only for slowly-changing sensor values. Finally, in SRT queries are issued from a fixed node while in DIM queries can be issued from any node.

A similar differentiation applies with respect to work on data-centric routing in sensor networks [15, 25], where data generated at a node is assumed to be stored at the node, and queries are either flooded throughout the network [15], or each source sets up a network-wide overlay announcing its presence so that mobile sinks can rendezvous with sources at the nearest node on the overlay [25]. These approaches work well for relatively long-lived queries.

Finally, our work is most close related to data-centric storage [23] systems, which include geographic hash-tables (GHTs) [20], DIMENSIONS [7], and DIFS [10].In a GHT, data is hashed by name to a location within the network, enabling highly efficient rendezvous. GHTs are built upon the GPSR [16] protocol and leverage some interesting properties of that protocol, such as the ability to route to a node nearest to a given location. We also leverage properties in GPSR (as we describe later), but we use a locality-preserving hash to store data, enabling efficient multi-dimensional range queries. DIMENSIONS and DIFS can be thought of as using the same set of primitives as GHT (storage using consistent hashing), but for different ends: DIMENSIONS allows drill-down search for features within a sensor network, while DIFS allows range queries on a single key in addition to other operations.

3. THE DESIGN OF DIMS

Most sensor networks are deployed to collect data from the environment. In these networks, nodes (either individually or collaboratively) will generate *events*. An event can generally be described as a tuple of attribute values, $\langle A_1, A_2, \cdots, A_k \rangle$, where each attribute A_i represents a sensor reading, or some value corresponding to a detection (*e.g.*, a confidence level). The focus of this paper is the design of systems to efficiently answer *multi-dimensional range queries* of the form: $\langle x_1 - y_1, x_2 - y_2, \cdots, x_k - y_k \rangle$. Such a query returns all events whose attribute values fall into the corresponding ranges. Notice that *point queries*, *i.e.,* queries that ask for events with specified values for each attribute, are a special case of range queries.

As we have discussed in Section 1, range queries can enable efficient *correlation* and triggering within the network. It is possible to implement range queries by flooding a query within the network. However, as we show in later sections, this alternative can be inefficient, particularly as the system scales, and if nodes within the network issue such queries relatively frequently. The other alternative, sending all events to an external storage node results in the access link being a bottleneck, especially if nodes within the network issue queries. Shenker *et al.* [23] also make similar arguments with respect to data-centric storage schemes in general; DIMs are an instance of such schemes.

The system we present in this paper, the DIM, relies upon two foundations: a locality-preserving geographic hash, and an underlying geographic routing scheme.

The key to resolving range queries efficiently is *data locality*: *i.e.,* events with comparable attribute values are stored nearby. The basic insight underlying DIM is that data locality can be obtained by a *locality-preserving geographic hash* function. Our geographic hash function finds a locality-preserving mapping from the multi-dimensional space (described by the set of attributes) to a 2-d geographic space; this mapping is inspired by k-d trees [2] and is described later. Moreover, each node in the network self-organizes to claim part of the attribute space for itself (we say that each node owns a *zone*), so events falling into that space are routed to and stored at that node.

Having established the mapping, and the zone structure, DIMs use a geographic routing algorithm previously developed in the literature to route events to their corresponding nodes, or to resolve queries. This algorithm, GPSR [16], essentially enables the delivery of a packet to a node at a specified location. The routing mechanism is simple: when a node receives a packet destined to a node at location X, it forwards the packet to the neighbor closest to X. In GPSR, this is called *greedy-mode forwarding.* When no such neighbor exists (as when there exists a *void* in the network), the node starts the packet on a *perimeter mode traversal*, using the well known right-hand rule to circumnavigate voids. GPSR includes efficient techniques for perimeter traversal that are based on graph planarization algorithms amenable to distributed implementation.

For all of this to work, DIMs make two assumptions that are consistent with the literature [23]. First, all nodes know the approximate geographic *boundaries* of the network. These boundaries may either be configured in nodes at the time of deployment, or may be discovered using a simple protocol. Second, each node knows its geographic location. Node locations can be automatically determined by a localization system or by other means.

Although the basic idea of DIMs may seem straightforward, it is challenging to design a completely distributed data structure that must be robust to packet losses and node failures, yet must support efficient query distribution and deal with communication voids and obstacles. We now describe the complete design of DIMs.

3.1 Zones

The key idea behind DIMs, as we have discussed, is a geographic locality-preserving hash that maps a multi-attribute event to a geographic *zone*. Intuitively, a zone is a subdivision of the geographic extent of a sensor field.

A zone is defined by the following constructive procedure. Consider a rectangle R on the x-y plane. Intuitively, R is the bounding rectangle that contains all sensors withing the network. We call a sub-rectangle Z of R a *zone*, if Z is obtained by dividing R k times, $k \geq 0$, using a procedure that satisfies the following property:

> After the i-th division, $0 \leq i \leq k$, R is partitioned into 2^i equal sized rectangles. If i is an odd (even) number, the i-th division is parallel to the y-axis (x-axis).

That is, the bounding rectangle R is first sub-divided into two zones at level 0 by a vertical line that splits R into two equal pieces, each of these sub-zones can be split into two zones at level 1 by a horizontal line, and so on. We call the non-negative integer k the level of zone Z, *i.e.* $level(Z) = k$.

A zone can be identified either by a zone code $code(Z)$ or by an address $addr(Z)$. The code $code(Z)$ is a 0-1 bit string of length $level(Z)$, and is defined as follows. If Z lies in the *left* half of R, the first (from the left) bit of $code(Z)$ is 0, else 1. If Z lies in the *bottom* half of R, the second bit of $code(Z)$ is 0, else 1. The remaining bits of $code(Z)$ are then recursively defined on each of the four quadrants of R. This definition of the zone code matches the definition of zones given above, encoding divisions of the sensor field geography by bit strings. Thus, in Figure 2, the zone in the top-right corner of the rectangle R has a zone code of 1111. Note that the zone codes collectively define a *zone tree* such that individual zones are at the leaves of this tree.

The address of a zone Z, $addr(Z)$, is defined to be the centroid of the rectangle defined by Z. The two representations of a zone (its code and its address) can each be computed from the other, assuming the level of the zone is known.

Two zones are called *sibling zones* if their zone codes are the same except for the last bit. For example, if $code(Z_1) =$ 01101 and $code(Z_2) =$ 01100, then Z_1 and Z_2 are sibling zones. The *sibling subtree* of a zone is the subtree rooted at the left or right sibling of the zone in the zone tree. We uniquely define a *backup zone* for each zone as follows: if the sibling subtree of the zone is on the left, the backup zone is the right-most zone in the sibling subtree; otherwise, the backup zone is the left-most zone in the sibling subtree. For a zone Z, let p be the first $level(Z) - 1$ digits of $code(Z)$. Let $backup(Z)$ be the backup zone of zone Z. If $code(Z) = p1$, $code(backup(Z)) = p01*$ with the most number of trailing 1's ($*$ means 0 or 1 occurrences). If

$code(Z) = p0$, $code(backup(Z)) = p10*$ with the most number of trailing 0's.

3.2 Associating Zones with Nodes

Our definition of a zone is independent of the actual distribution of nodes in the sensor field, and only depends upon the geographic extent (the bounding rectangle) of the sensor field. Now we describe how zones are mapped to nodes.

Conceptually, the sensor field is logically divided into zones and each zone is assigned to a single node. If the sensor network were deployed in a grid-like (*i.e.,* very regular) fashion, then it is easy to see that there exists a k such that each node maps into a distinct level-k zone. In general, however, the node placements within a sensor field are likely to be less regular than the grid. For some k, some zones may be empty and other zones might have more than one node situated within them. One alternative would have been to choose a fixed k for the overall system, and then associate nodes with the zones they are in (and if a zone is empty, associate the "nearest" node with it, for some definition of "nearest"). Because it makes our overall query routing system simpler, we allow nodes in a DIM to map to different-sized zones.

To precisely understand the associations between zones and nodes, we define the notion of zone *ownership.* For any given placement of network nodes, consider a node A. Let Z_A to be the largest zone that includes only node A and no other node. Then, we say that A *owns* Z_A. Notice that this definition of ownership may leave some sections of the sensor field un-associated with a node. For example, in Figure 2, the zone 110 does not contain any nodes and would not have an owner. To remedy this, for any empty zone Z, we define the owner to be the owner of *backup*(Z). In our example, that empty zone's owner would also be the node that owns 1110, its backup zone.

Having defined the association between nodes and zones, the next problem we tackle is: given a node placement, does there exist a distributed algorithm that enables each node to determine which zones it owns, knowing only the overall boundary of the sensor network? In principle, this should be relatively straightforward, since each node can simply determine the location of its neighbors, and apply simple geometric methods to determine the largest zone around it such that no other node resides in that zone. In practice, however, communication voids and obstacles make the algorithm much more challenging. In particular, resolving the ownership of zones that do not contain any nodes is complicated. Equally complicated is the case where the zone of a node is larger than its communication radius and the node cannot determine the boundaries of its zone by local communication alone.

Our distributed zone building algorithm *defers* the resolution of such zones until when either a query is initiated, or when an event is inserted. The basic idea behind our algorithm is that each node tentatively builds up an idea of the zone it resides in just by communicating with its neighbors (remembering which boundaries of the zone are "undecided" because there is no radio neighbor that can help resolve that boundary). These undecided boundaries are later resolved by a GPSR perimeter traversal when data messages are actually routed.

We now describe the algorithm, and illustrate it using examples. In our algorithm, each node uses an array *bound*[0..3] to maintain the four boundaries of the zone it owns (remember that in this algorithm, the node only tries to determine the zone it resides in, not the other zones it might own because those zones are devoid of nodes). When a node starts up, each node initializes this array to be the network boundary, *i.e.,* initially each node assumes its zone contains the whole network. The zone boundary algorithm now relies upon GPSR's beacon messages to learn the locations of neighbors within radio range. Upon hearing of such a neighbor, the node calls the algorithm in Figure 4 to update its zone boundaries and its code accordingly. In this algorithm, we assume that A is the node at which the algorithm is executed, Z_A is its zone, and a is a newly discovered neighbor of A. (Procedure CONTAIN(Z_A, a) is used to decide if node a is located within the current zone boundaries of node A).

Using this algorithm, then, each node can independently and asynchronously decide its own tentative zone based on the location of its neighbors. Figure 2 illustrates the results of applying this algorithm for the network in Figure 1.

Figure 3 describes the corresponding zone tree. Each zone resides at a leaf node and the code of a zone is the path from the root to the zone if we represent the branch to the left

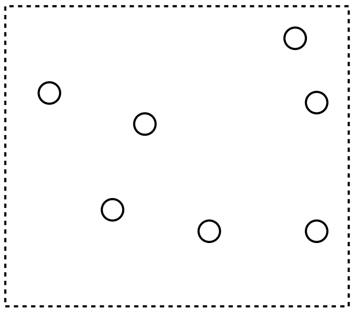

Figure 1: A network, where circles represent sensor nodes and dashed lines mark the network boundary.

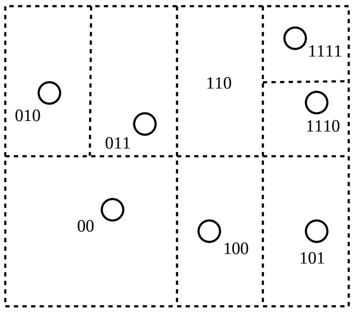

Figure 2: The zone code and boundaries.

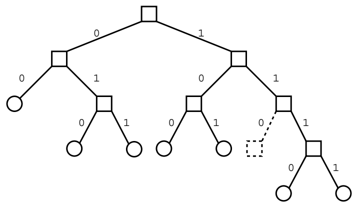

Figure 3: The Corresponding Zone Tree

```
BUILD-ZONE($a$)
1  while CONTAIN($Z_A, a$)
2  do if $length(code(Z_A))$ mod $2 = 0$
3        then $new\_bound \leftarrow (bound[0] + bound[1])/2$
4             if $A.x < new\_bound$
5                then $bound[1] \leftarrow new\_bound$
6                else $bound[0] \leftarrow new\_bound$
7        else $new\_bound \leftarrow (bound[2] + bound[3])/2$
8             if $A.y < new\_bound$
9                then $bound[3] \leftarrow new\_bound$
10               else $bound[2] \leftarrow new\_bound$
11     Update zone code $code(Z_A)$
```

Figure 4: Zone Boundary Determination, where $A.x$ and $A.y$ represent the geographic coordinate of node A.

```
INSERT-EVENT($e$)
1  $c \leftarrow$ ENCODE($e$)
2  if CONTAIN($Z_A, c$) $= true$ and IS_INTERNAL() $= true$
3     then Store $e$ and exit
4  SEND-MESSAGE($c, e$)

SEND-MESSAGE($c, m$)
1  if $\exists$ neighbor $Y$, CLOSER($Y, owner(m), m$) $= true$
2     then $addr(m) \leftarrow addr(Y)$
3     else if $length(c) > length(code(m))$
4             then Update $code(m)$ and $addr(m)$
5                  $source(m) \leftarrow$ caller
6          if IS_OWNER($msg$) $= true$
7             then $owner(m) \leftarrow$ caller's code
8  SEND($m$)
```

Figure 5: Inserting an event in a DIM. Procedure CLOSER(A, B, m) returns true if $code(A)$ is closer to $code(m)$ than $code(B)$. $source(m)$ is used to set the source address of message m.

child by 0 and the branch to the right child by 1. This binary tree forms the index that we will use in the following event and query processing procedures.

We see that the zone sizes are different and depend on the local densities and so are the lengths of zone codes for different nodes. Notice that in Figure 2, there is an empty zone whose code should be 110. In this case, if the node in zone 1111 can only hear the node in zone 1110, it sets its boundary with the empty zone to *undecided*, because it did not hear from any neighboring nodes from that direction. As we have mentioned before, the undecided boundaries are resolved using GPSR's perimeter mode when an event is inserted, or a query sent. We describe event insertion in the next step.

Finally, this description does not describe how a node's zone codes are adjusted when neighboring nodes fail, or new nodes come up. We return to this in Section 3.5.

3.3 Inserting an Event

In this section, we describe how events are inserted into a DIM. There are two algorithms of interest: a consistent hashing technique for mapping an event to a zone, and a routing algorithm for storing the event at the appropriate zone. As we shall see, these two algorithms are inter-related.

3.3.1 Hashing an Event to a Zone

In Section 3.1, we described a recursive tessellation of the geographic extent of a sensor field. We now describe a consistent hashing scheme for a DIM that supports range queries on m distinct attributes[2]

Let us denote these attributes $A_1 \ldots A_m$. For simplicity, assume for now that the depth of every zone in the network is k, k is a multiple of m, and that this value of k is known to every node. We will relax this assumption shortly. Furthermore, for ease of discussion, we assume that all attribute values have been normalized to be between 0 and 1.

Our hashing scheme assigns a k bit zone code to an event as follows. For i between 1 and m, if $A_i < 0.5$, the i-th bit of the zone code is assigned 0, else 1. For i between $m + 1$ and $2m$, if $A_{i-m} < 0.25$ or $A_{i-m} \in [0.5, 0.75)$, the i-th bit of the zone is assigned 0, else 1, because the next level divisions are at 0.25 and 0.75 which divide the ranges to $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$, and $[0.75, 1)$. We repeat this procedure until all k bits have been assigned. As an example, consider event $E = \langle 0.3, 0.8\rangle$. For this event, the 5-bit zone code is $code(Z_A) = 01110$.

Essentially, our hashing scheme uses the values of the attributes in round-robin fashion on the zone tree (such as the one in Figure 3), in order to map an m-attribute event to a zone code. This is reminiscent of k-d trees [2], but is quite different from that data structure: zone trees are spatial embeddings and do not incorporate the re-balancing algorithms in k-d trees.

In our design of DIMs, we do not require nodes to have zone codes of the same length, nor do we expect a node to know the zone codes of other nodes. Rather, suppose the encoding node is A and its own zone code is of length k_A. Then, given an event E, node A only hashes E to a zone code of length k_A. We denote the zone code assigned to an event E by $code(E)$. As we describe below, as the event is routed, $code(E)$ is refined by intermediate nodes. This *lazy evaluation* of zone codes allows different nodes to use different length zone codes without any explicit coordination.

3.3.2 Routing an Event to its Owner

The aim of hashing an event to a zone code is to store the event at the node within the network node that owns that zone. We call this node the *owner* of the event. Consider an event E that has just been generated at a node A. After encoding event E, node A compares $code(E)$ with $code(A)$. If the two are identical, node A store event E locally; otherwise, node A will attempt to *route* the event to its owner.

To do this, note that $code(E)$ corresponds to some zone Z', which is A's current guess for the zone at which event E should be stored. A now invokes GPSR to send a message to $addr(Z')$ (the centroid of Z', Section 3.1). The message contains the event E, $code(E)$, and the target geographic location for storing the event. In the message, A also marks itself as the *owner* of event E. As we will see later, the guessed zone Z', the address $addr(Z')$, and the owner of E, all of them contained in the message, will be refined by intermediate forwarding nodes.

GPSR now delivers this message to the next hop towards $addr(Z')$ from A. This next hop node (call it B) does not immediately forward the message. Rather, it attempts to com-

[2]DIM does not assume that all nodes are homogeneous in terms of the sensors they have. Thus, in an m dimensional DIM, a node that does not possess all m sensors can use `NULL` values for the corresponding readings. DIM treats `NULL` as an extreme value for range comparisons. As an aside, a network may have many DIM instances running concurrently.

pute a new zone code for E to get a new code $code_{new}(E)$. B will update the code contained in the message (and also the geographic destination of the message) if $code_{new}(E)$ is longer than the event code in the message. In this manner, as the event wends its way to its owner, its zone code gets refined. Now, B compares its own code $code(B)$ against the owner code $owner(E)$ contained in the incoming message. If $code(B)$ has a longer match with $code(E)$ than the current owner $owner(E)$, then B sets itself to be the current owner of E, meaning that if nobody is eligible to store E, then B will store the event (we shall see how this happens next). If B's zone code does not exactly match $code(E)$, B will invoke GPSR to deliver E to the next hop.

3.3.3 Resolving undecided zone boundaries during insertion

Suppose that some node, say C, finds itself to be the destination (or eventual owner) of an event E. It does so by noticing that code $code(C)$ equals $code(E)$ after locally recomputing a code for E. In that case, C stores E locally, but only if all four of C's zone boundaries are decided. When this condition holds, C knows for sure that no other nodes have overlapped zones with it. In this case, we call C an *internal node*.

Recall, though, that because the zone discovery algorithm Section 3.2 only uses information from immediate neighbors, one or more of C's boundaries may be *undecided*. If so, C assumes that some other nodes have a zone that overlaps with its own, and sets out to resolve this overlap. To do this, C now sets itself to be the owner of E and continues forwarding the message. Here we rely on GPSR's perimeter mode routing to probe around the void that causes the undecided boundary. Since the message starts from C and is destined for a geographic location near C, GPSR guarantees that the message will be delivered back to C if no other nodes will update the information in the message. If the message comes back to C with itself to be the owner, C infers that it must be the true owner of the zone and stores E locally.

If this does not happen, there are two possibilities. The first is that as the event traverses the perimeter, some intermediate node, say B whose zone overlaps with C's marks itself to be the owner of the event, but otherwise does not change the event's zone code. This node also recognizes that its own zone overlaps with C's and initiates a message exchange which causes each of them to appropriately shrink their zone.

Figures 6 through 8 show an example of this data-driven zone shrinking. Initially, both node A and node B have claimed the same zone 0 because they are out of radio range of each other. Suppose that A inserts an event $E = \langle 0.4, 0.8, 0.9 \rangle$. A encodes E to 0 and claims itself to be the owner of E. Since A is not an internal node, it sends out E, looking for other owner candidates of E. Once E gets to node B, B will see in the message's owner field A's code that is the same as its own. B then shrinks its zone from 0 to 01 according to A's location which is also recorded in the message and send a shrink request to A. Upon receiving this request, A also shrinks its zone from 0 to 00.

A second possibility is if some intermediate node changes the destination code of E to a more specific value (*i.e.,* longer zone code). Let us label this node D. D now tries to initiate delivery to the centroid of the new zone. This might result in a new perimeter walk that returns to D (if, for example, D happens to be geographically closest to the centroid of the zone). However, D would not be the owner of the event, which would still be C. In routing to the centroid of this zone, the message may traverse the perimeter and return to D. Now D notices that C was the original owner, so it encapsulates the event and directs it to C. In case that there indeed is another node, say X, that owns an overlapped zone with C, X will notice this fact by finding in the message the same prefix of the code of one of its zones, but with a different geographic location from its own. X will shrink its zone to resolve the overlap. If X's zone is smaller than or equal to C's zone, X will also send a "shrink" request to C. Once C receives a shrink request, it will reduce its zone appropriately and fix its "undecided" boundary. In this manner, the zone formation process is resolved on demand in a data-driven way.

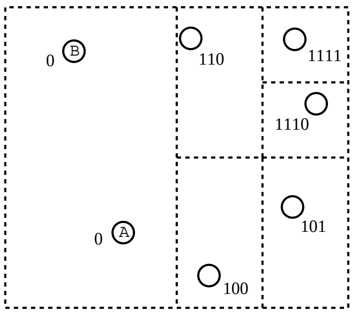

Figure 6: Nodes A and B have claimed the same zone.

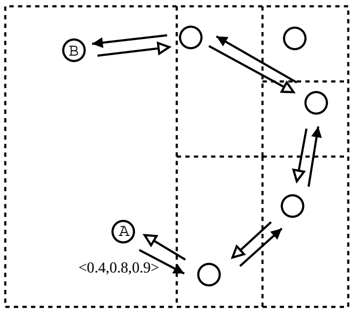

Figure 7: An event/query message (filled arrows) triggers zone shrinking (hollow arrows).

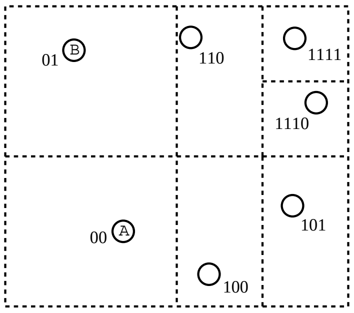

Figure 8: The zone layout after shrinking. Now node A and B have been mapped to different zones.

There are several interesting effects with respect to perimeter walking that arise in our algorithm. The first is that there are some cases where an event insertion might cause the entire outer perimeter of the network to be traversed[3]. Figure 6 also works as an example where the outer perimeter is traversed. Event E inserted by A will eventually be stored in node B. Before node B stores event E, if B's nominal radio range does not intersect the network boundary, it needs to send out E again as A did, because B in this case is not an internal node. But if B's nominal radio range intersects the network boundary, it then has two choices. It can assume that there will not be any nodes outside the network boundary and so B is an internal node. This is an aggressive approach. On the other hand, B can also make a conservative decision assuming that there might be some other nodes it have not heard of yet. B will then force the message walking another perimeter before storing it.

In some situations, especially for large zones where the node that owns a zone is far away from the centroid of the owned zone, there might exist a small perimeter around the destination that does not include the owner of the zone. The event will end up being stored at a different node than the real owner. In order to deal with this problem, we add an extra operation in event forwarding, called *efficient neighbor discovery*. Before invoking GPSR, a node needs to check if there exists a *neighbor* who is eligible to be the real owner of the event. To do this, a node C, say, needs to know the zone codes of its neighboring nodes. We deploy GPSR's beaconing message to piggyback the zone codes for nodes. So by simply comparing the event's code and neighbor's code, a node can decide whether there exists a neighbor Y which is more likely to be the owner of event E. C delivers E to Y, which simply follows the decision making procedure discussed above.

3.3.4 Summary and Pseudo-code

In summary, our event insertion procedure is designed to nicely interact with the zone discovery mechanism, and the event hashing mechanism. The latter two mechanisms are kept simple, while the event insertion mechanism uses lazy evaluation at each hop to refine the event's zone code, and it leverages GPSR's perimeter walking mechanism to fix undecided zone boundaries. In Section 3.5, we address robustness of event insertion to packet loss or to node failures.

Figure 5 shows the pseudo-code for inserting and forwarding an event e. In this pseudo code, we have omitted a description of the zone shrinking procedure. In the pseudo code, procedure IS_INTERNAL() is used to determine if the caller is an internal node and procedure IS_OWNER() is used to determine if the caller is more eligible to be the owner of the event than is currently claimed owner as recorded in the message. Procedure SEND-MESSAGE is used to send either an event message or a query message. If the message destination address has been changed, the packet source address needs also to be changed in order to avoid being dropped by GPSR, since GPSR does not allow a node to see the same packet in greedy mode twice.

[3]This happens less frequently than for GHTs, where inserting an event to a location outside the actual (but inside the nominal) boundary of the network will always invoke an external perimeter walk.

3.4 Resolving and Routing Queries

DIMs support both point queries[4] and range queries. Routing a point query is identical to routing an event. Thus, the rest of this section details how range queries are routed.

The key challenge in routing zone queries is brought out by the following strawman design. If the entire network was divided evenly into zones of depth k (for some pre-defined constant k), then the querier (the node issuing the query) could subdivide a given range query into the relevant subzones and route individual requests to each of the zones. This can be inefficient for large range queries and also hard to implement in our design where zone sizes are not predefined. Accordingly, we use a slightly different technique where a range query is initially routed to a zone corresponding to the entire range, and is then progressively *split* into smaller subqueries. We describe this algorithm here.

The first step of the algorithm is to map a range query to a zone code prefix. Conceptually, this is easy; in a zone tree (Figure 3), there exists some node which contains the entire range query in its sub-tree, and none of its children in the tree do. The initial zone code we choose for the query is the zone code corresponding to that tree node, and is a prefix of the zone codes of all zones (note that these zones may not be geographically contiguous) in the subtree. The querier computes the zone code of Q, denoted by $code(Q)$ and then starts routing a query to $addr(code(Q))$.

Upon receiving a range query Q, a node A (where A is any node on the query propagation path) divides it into multiple smaller sized *subqueries* if there is an overlap between the zone of A, $zone(A)$ and the zone code associated with Q, $code(Q)$. Our approach to split a query Q into subqueries is as follows. If the range of Q's first attribute contains the value 0.5, A divides Q into two sub-queries one of whose first attribute ranges from 0 to 0.5, and the other from 0.5 to 1. Then A decides the half that overlaps with its own zone. Let's call it Q_A. If Q_A does not exist, then A stops splitting; otherwise, it continues splitting (using the second attribute range) and recomputing Q_A until Q_A is small enough so that it completely falls into $zone(A)$ and hence A can now resolve it. For example, suppose that node A, whose code is 0110, is to split a range query $Q = \langle 0.3 - 0.8, 0.6 - 0.9\rangle$. The splitting steps is shown in Figure 2. After splitting, we obtain three smaller queries $q_0 = \langle 0.3 - 0.5, 0.6 - 0.75\rangle$, $q_1 = \langle 0.3 - 0.5, 0.75 - 0.9\rangle$, and $q_2 = \langle 0.5 - 0.8, 0.6 - 0.9\rangle$.

This splitting procedure is illustrated in Figure 9 which also shows the codes of each subquery after splitting.

A then replies to subquery q_0 with data stored locally and sends subqueries q_1 and q_2 using the procedure outlined above. More generally, if node A finds itself to be inside the zone subtree that maximally covers Q, it will send the subqueries that resulted from the split. Otherwise, if there is no overlap between A and Q, then A forwards Q as is (in this case Q is either the original query, or a product of an earlier split).

Figure 10 describes the pseudo-code for the zone splitting algorithm. As shown in the above algorithm, once a subquery has been recognized as belonging to the caller's zone, procedure RESOLVE is invoked to resolve the subquery and send a reply to the querier. Every query message contains

[4]By point queries, we mean the equality condition on all indexed keys. DIM index attributes are not necessarily primary keys.

the geographic location of its initiator, so the corresponding reply message can be delivered directly back to the initiator. Finally, in the process of query resolution, zones might shrink similar to shrinkage during inserting. We omit this in the pseudo code.

3.5 Robustness

Until now, we have not discussed the impact of node failures and packet losses, or node arrivals and departures on our algorithms. Packet losses can affect query and event insertion, and node failures can result in lost data, while node arrivals and departures can impact the zone structure. We now discuss how DIMs can be made robust to these kinds of dynamics.

3.5.1 Maintaining Zones

In previous sections, we described how the zone discovery algorithm could leave zone boundaries undecided. These undecided boundaries are resolved during insertion or querying, using the zone shrinking procedure describe above.

When a new node joins the network, the zone discovery mechanism (Section 3.2) will cause neighboring zones to appropriately adjust their zone boundaries. At this time, those zones can also transfer to the new node those events they store but which should belong to the new node.

Before a node turns itself off (if this is indeed possible), it knows that its backup node (Section 3.1) will take over its zone, and will simply send all its events to its backup node. Node deletion may also cause *zone expansion*. In order to keep the mapping between the binary zone tree's leaf nodes and zones, we allow zone expansion to only occur among sibling zones (Section 3.1). The rule is: if $zone(A)$'s sibling zone becomes empty, then A can expand its own zone to include its sibling zone.

Now, we turn our attention to node failures. Node failures are just like node deletions except that a failed node does not have a chance to move its events to another node. But how does a node decide if its sibling has failed? If the sibling is within radio range, the absence of GPSR beaconing messages can detect this. Once it detects this, the node can expand its zone. A different approach is needed for detecting siblings who are not within radio range. These are the cases where two nodes own their zones after exchanging a shrink message; they do not periodically exchange messages thereafter to maintain this zone relationship. In this case, we detect the failure in a data-driven fashion, with obvious efficiency benefits compared to periodic keepalives. Once a node B has failed, an event or query message that previously should have been owned by the failed node will now be delivered to the node A that owns the empty zone left by node B. A can see this message because A stands right around the empty area left by B and is guaranteed to be visited in a GPSR perimeter traversal. A will set itself to be the owner of the message, and any node which would have dropped this message due to a perimeter loop will redirect the message to A instead. If A's zone happens to be the sibling of B's zone, A can safely expand its own zone and notify its expanded zone to its neighbors via GPSR beaconing messages.

3.5.2 Preventing Data Loss from Node Failure

The algorithms described above are robust in terms of zone formation, but node failure can erase data. To avoid this, DIMs can employ two kinds of *replication*: *local* replication to be resilient to random node failures, and *mirror* replication for resilience to concurrent failure of geographically contiguous nodes.

Mirror replication is conceptually easy. Suppose an event E has a zone code $code(E)$. Then, the node that inserts E would store two copies of E; one at the zone denoted by $code(E)$, and the other at the zone corresponding to the *one's complement* of $code(E)$. This technique essentially creates a mirror DIM. A querier would need, in parallel, to query both the original DIM and its mirror since there is no way of knowing if a collection of nodes has failed. Clearly, the trade-off here is an approximate doubling of both insertion and query costs.

There exists a far cheaper technique to ensure resilience to random node failures. Our local replication technique rests on the observation that, for each node A, there exists a unique node which will take over its zone when A fails. This node is defined as the node responsible for A's zone's backup zone (see Section 3.1). The basic idea is that A replicates each data item it has in this node. We call this node A's *local replica*. Let A's local replica be B. Often B will be a radio neighbor of A and can be detected from GPSR beacons. Sometimes, however, this is not the case, and B will have to be explicitly discovered.

We use an explicit message for discovering the local replica. Discovering the local replica is data-driven, and uses a mechanism similar to that of event insertion. Node A sends a message whose geographic destination is a random nearby location chosen by A. The location is close enough to A such that GPSR will guarantee that the message will delivered back to A. In addition, the message has three fields, one for the zone code of A, $code(A)$, one for the owner $owner(A)$ of $zone(A)$ which is *set to be empty*, and one for the geographic location of $owner(A)$. Then the packet will be delivered in GPSR perimeter mode. Each node that receives this message will compare its zone code and $code(A)$ in the message, and if it is more eligible to be the owner of $zone(A)$ than the current $owner(A)$ recorded in the message, it will update the field $owner(A)$ and the corresponding geographic location. Once the packet comes back to A, it will know the location of its local replica and can start to send replicas.

In a dense sensor network, the local replica of a node is usually very near to the node, either its direct neighbor or 1–2 hops away, so the cost of sending replicas to local replication will not dominate the network traffic. However, a node's local replica itself may fail. There are two ways to deal with this situation; periodic refreshes, or repeated data-driven discovery of local replicas. The former has higher overhead, but more quickly discovers failed replicas.

3.5.3 Robustness to Packet Loss

Finally, the mechanisms for querying and event insertion can be easily made resilient to packet loss. For event insertion, a simple ACK scheme suffices.

Of course, queries and responses can be lost as well. In this case, there exists an efficient approach for error recovery. This rests on the observation that the querier knows which zones fall within its query and should have responded (we assume that a node that has no data matching a query, but whose zone falls within the query, responds with a negative acknowledgment). After a conservative timeout, the querier can re-issue the queries *selectively* to these zones. If DIM cannot get any answers (positive or negative) from

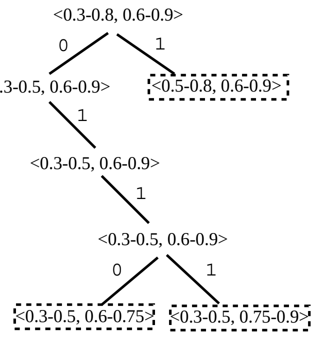

Figure 9: An example of range query splitting

```
RESOLVE-RANGE-QUERY($Q$)
 1  $Q_{sub} \leftarrow$ NIL
 2  $q_0, Q_{sub} \leftarrow$ SPLIT-QUERY($Q$)
 3  if $q_0 =$ NIL
 4     then $c \leftarrow$ ENCODE($Q$)
 5          if CONTAIN($c, code(A)$) $= true$
 6             then go to step 12
 7             else SEND-MESSAGE($c, q_0$)
 8     else RESOLVE($q_0$)
 9          if IS_INTERNAL() $= true$
10             then Absorb ($q_0$)
11             else Append $q_0$ to $Q_{sub}$
12          if $Q_{sub} \neq$ NIL
13             then for each subquery $q \in Q_{sub}$
14                     do $c \leftarrow$ ENCODE($q$)
15                        SEND-MESSAGE($c, q$)
```

Figure 10: Query resolving algorithm

certain zones after repeated timeouts, it can at least return the partial query results to the application together with the information about the zones from which data is missing.

4. DIMS: AN ANALYSIS

In this section, we present a simple analytic performance evaluation of DIMs, and compare their performance against other possible approaches for implementing multi-dimensional range queries in sensor networks. In the next section, we validate these analyses using detailed packet-level simulations.

Our primary metrics for the performance of a DIM are:

Average Insertion Cost measures the average number of messages required to insert an event into the network.

Average Query Delivery Cost measures the average number of messages required to route a query message to all the relevant nodes in the network.

It does not measure the number of messages required to transmit responses to the querier; this latter number depends upon the precise data distribution and is the same for many of the schemes we compare DIMs against.

In DIMs, event insertion essentially uses geographic routing. In a dense N-node network where the likelihood of traversing perimeters is small, the average event insertion cost proportional to $\sqrt{N}$ [23].

On the other hand, the query delivery cost depends upon the size of ranges specified in the query. Recall that our query delivery mechanism is careful about splitting a query into sub-queries, doing so only when the query nears the zone that covers the query range. Thus, when the querier is far from the queried zone, there are two components to the query delivery cost. The first, which is proportional to $\sqrt{N}$, is the cost to deliver the query near the covering zone. If within this covering zone, there are M nodes, the message delivery cost of splitting the query is proportional to M.

The *average* cost of query delivery depends upon the distribution of query range sizes. Now, suppose that query sizes follow some density function $f(x)$, then the average cost of resolve a query can be approximated by $\int_1^N xf(x)dx$. To give some intuition for the performance of DIMs, we consider four different forms for $f(x)$: the *uniform* distribution where a query range encompassing the entire network is as likely as a point query; a *bounded uniform* distribution where all sizes up to a bound B are equally likely; an *algebraic* distribution in which most queries are small, but large queries are somewhat likely; and an *exponential* distribution where most queries are small and large queries are unlikely. In all our analyses, we make the simplifying assumption that the size of a query is proportional to the number of nodes that can answer that query.

For the uniform distribution $P(x) \propto c$ for some constant c. If each query size from $1 \ldots N$ is equally likely, the average query delivery cost of uniformly distributed queries is $O(N)$. Thus, for uniformly distributed queries, the performance of DIMs is comparable to that of flooding. However, for the applications we envision, where nodes within the network are trying to correlate events, the uniform distribution is highly unrealistic.

Somewhat more realistic is a situation where all query sizes are bounded by a constant B. In this case, the average cost for resolving such a query is approximately $\int_1^B xf(x)dx = O(B)$. Recall now that all queries have to pay an approximate cost of $O(\sqrt{N})$ to deliver the query near the covering zone. Thus, if DIM limited queries to a size proportional to $\sqrt{N}$, the average query cost would be $O(\sqrt{N})$.

The algebraic distribution, where $f(x) \propto x^{-k}$, for some constant k between 1 and 2, has an average query resolution cost given by $\int_1^N xf(x)dx = O(N^{2-k})$. In this case, if $k > 1.5$, the average cost of query delivery is dominated by the cost to deliver the query to near the covering zone, given by $O(\sqrt{N})$.

Finally, for the exponential distribution, $f(x) = ce^{-cx}$ for some constant c, and the average cost is just the mean of the corresponding distribution, *i.e.,* $O(1)$ for large N. Asymptotically, then, the cost of the query for the exponential distribution is dominated by the cost to deliver the query near the covering zone ($O(\sqrt{N})$).

Thus, we see that if queries follow either the bounded uniform distribution, the algebraic distribution, or the exponential distribution, the query cost scales as the insertion cost (for appropriate choice of constants for the bounded uniform and the algebraic distributions).

How well does the performance of DIMs compare against alternative choices for implementing multi-dimensional queries? A simple alternative is called *external storage* [23], where all events are stored centrally in a node outside the sensor network. This scheme incurs an insertion cost of $O(\sqrt{N})$, and a zero query cost. However, as [23] points out, such systems may be impractical in sensor networks since the access link to the external node becomes a *hotspot.*

A second alternative implementation would store events at the node where they are generated. Queries are *flooded* throughout the network, and nodes that have matching data respond. Examples of systems that can be used for this (although, to our knowledge, these systems do not implement multi-dimensional range queries) are Directed Diffusion [15] and TinyDB [17]. The flooding scheme incurs a zero insertion cost, but an $O(N)$ query cost. It is easy to show that DIMs outperform flooding as long as the ratio of the number of insertions to the number of queries is less than $\sqrt{N}$.

A final alternative would be to use a geographic hash table (GHT [20]). In this approach, attribute values are assumed to be integers (this is actually quite a reasonable assumption since attribute values are often quantized), and events are hashed on some (say, the first) attribute. A range query is sub-divided into several sub-queries, one for each integer in the range of the first attribute. Each sub-query is then hashed to the appropriate location. The nodes that receive a sub-query only return events that match all other attribute ranges. In this approach, which we call GHT-R (GHT's for range queries) the insertion cost is $O(\sqrt{N})$. Suppose that the range of the first attribute contains r discrete values. Then the cost to deliver queries is $O(r\sqrt{N})$. Thus, asymptotically, GHT-R's perform similarly to DIMs. In practice, however, the proportionality constants are significantly different, and DIMs outperform GHT-Rs, as we shall show using detailed simulations.

5. DIMS: SIMULATION RESULTS

Our analysis gives us some insight into the asymptotic behavior of various approaches for multi-dimensional range queries. In this section, we use simulation to compare DIMs against flooding and GHT-R; this comparison gives us a more detailed understanding of these approaches for moderate size networks, and gives us a nuanced view of the mechanistic differences between some of these approaches.

5.1 Simulation Methodology

We use *ns-2* for our simulations. Since DIMs are implemented on top of GPSR, we first ported an earlier GPSR implementation to the latest version of *ns-2*. We modified the GPSR module to call our DIM implementation when it receives any data message in transit or when it is about to drop a message because that message traversed the entire perimeter. This allows a DIM to modify message zone codes in flight (Section 3), and determine the actual owner of an event or query.

In addition, to this, we implemented in *ns-2* most of the DIM mechanisms described in Section 3. Of those mechanisms, the only one we did not implement is mirror replication. We have implemented selective query retransmission for resiliency to packet loss, but have left the evaluation of this mechanism to future work. Our DIM implementation in *ns-2* is 2800 lines of code.

Finally, we implemented GHT-R, our GHT-based multi-dimensional range query mechanism in *ns-2*. This implementation was relatively straightforward, given that we had ported GPSR, and modified GPSR to detect the completion of perimeter mode traversals.

Using this implementation, we conducted a fairly extensive evaluation of DIM and two alternatives (flooding, and our GHT-R). For all our experiments, we use uniformly placed sensor nodes with network sizes ranging from 50 nodes to 300 nodes. Each node has a radio range of 40m. For the results presented here, each node has on average 20 nodes within its nominal radio range. We have conducted experiments at other node densities; they are in agreement with the results presented here.

In all our experiments, each node first generates 3 events[5] on average (more precisely, for a topology of size N, we have $3N$ events, and each node is equally likely to generate an event). We have conducted experiments for three different event value distributions. Our *uniform* event distribution generates 2-dimensional events and, for each dimension, every attribute value is equally likely. Our *normal* event distribution generates 2-dimensional events and, for each dimension, the attribute value is normally distributed with a mean corresponding to the mid-point of the attribute value range. The normal event distribution represents a skewed data set. Finally, our *trace* event distribution is a collection of 4-dimensional events obtained from a habitat monitoring network. As we shall see, this represents a fairly skewed data set.

Having generated events, for each simulation we generate queries such that, on average, each node generates 2 queries. The query sizes are determined using the four size distributions we discussed in Section 4: uniform, bounded-uniform, algebraic and exponential. Once a query size has been determined, the location of the query (*i.e.,* the actual boundaries of the zone) are uniformly distributed. For our GHT-R experiments, the dynamic range of the attributes had 100 discrete values, but we restricted the query range for any one attribute to 50 discrete values to allow those simulations to complete in reasonable time.

Finally, using one set of simulations we evaluate the efficacy of local replication by turning off random fractions of nodes and measuring the fidelity of the returned results.

The primary metrics for our simulations are the average query and insertion costs, as defined in Section 4.

5.2 Results

Although we have examined almost all the combinations of factors described above, we discuss only the most salient ones here, for lack of space.

Figure 11 plots the average insertion costs for DIM and GHT-R (for flooding, of course, the insertion costs are zero). DIM incurs less per event overhead in inserting events (regardless of the actual event distribution; Figure 11 shows the cost for uniformly distributed events). The reason for this is interesting. In GHT-R, storing almost every event incurs a perimeter traversal, and storing some events require traversing the outer perimeter of the network [20]. By contrast, in DIM, storing an event incurs a perimeter traversal only when a node's boundaries are undecided. Furthermore, an insertion or a query in a DIM can traverse the outer perimeter (Section 3.3), but less frequently than in GHTs.

Figure 13 plots the average query cost for a bounded uniform query size distribution. For this graph (and the next) we use a uniform event distribution, since the event distribution does not affect the query delivery cost. For this simulation, our bound was $\frac{1}{4}$th the size of the largest possible

[5]Our metrics are chosen so that the exact number of events and queries is unimportant for our discussion. Of course, the overall performance of the system will depend on the relative frequency of events and queries, as we discuss in Section 4. Since we don't have realistic ratios for these, we focus on the microscopic costs, rather than on the overall system costs.

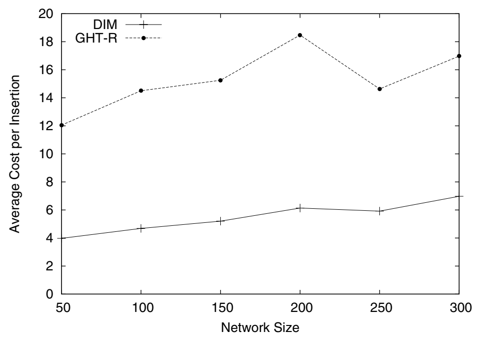

Figure 11: Average insertion cost for DIM and GHT.

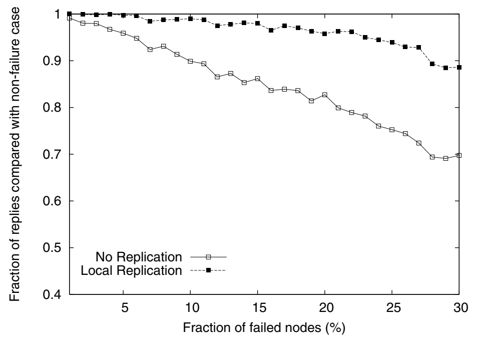

Figure 12: Local replication performance.

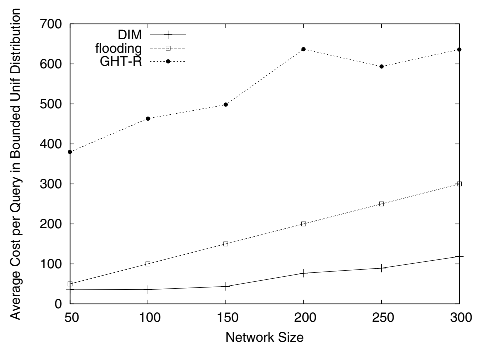

Figure 13: Average query cost with a bounded uniform query distribution

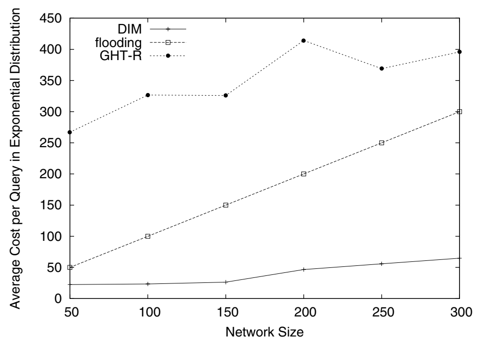

Figure 14: Average query cost with an exponential query distribution

query (*e.g.*, a query of the form $\langle 0-0.5, 0-0.5\rangle$. Even for this generous query size, DIMs perform quite well (almost a third the cost of flooding). Notice, however, that GHT-Rs incur high query cost since almost any query requires as many subqueries as the width of the first attribute's range.

Figure 14 plots the average query cost for the exponential distribution (the average query size for this distribution was set to be $\frac{1}{16}$th the largest possible query). The superior scaling of DIMs is evident in these graphs. Clearly, this is the regime in which one might expect DIMs to perform best, when most of the queries are small and large queries are relatively rare. This is also the regime in which one would expect to use multi-dimensional range queries: to perform relatively tight correlations. As with the bounded uniform distribution, GHT query cost is dominated by the cost of sending sub-queries; for DIMs, the query splitting strategy works quite well in keep overall query delivery costs low.

Figure 12 describes the efficacy of local replication. To obtain this figure, we conducted the following experiment. On a 100-node network, we inserted a number of events uniformly distributed throughout the network, then issued a query covering the entire network and recorded the answers. Knowing the expected answers for this query, we then successively removed a fraction f of nodes randomly, and re-issued the same query. The figure plots the fraction of expected responses actually received, with and without replication. As the graph shows, local replication performs well for random failures, returning almost 90% of the responses when up to 30% of the nodes have failed *simultaneously* [6].In the absence of local replication, of course, when 30% of the nodes fail, the response rate is only 70% as one would expect.

We note that DIMs (as currently designed) are not perfect. When the data is highly skewed—as it was for our trace data set from the habitat monitoring application where most of the event values fell into within 10% of the attribute's range—a few DIM nodes will clearly become the bottleneck. This is depicted in Figure 15, which shows that for DIMs, and GHT-Rs, the maximum number of transmissions at any network node (the hotspots) is rather high. (For less skewed data distributions, and reasonable query size distributions, the hotspot curves for all three schemes are comparable.) This is a standard problem that the database indices have dealt with by tree re-balancing. In our case, simpler solutions might be possible (and we discuss this in Section 7).

However, our use of the trace data demonstrates that DIMs work for events which have more than two dimensions. Increasing the number of dimensions does not noticeably degrade DIMs query cost (omitted for lack of space).

Also omitted are experiments examining the impact of several other factors, as they do not affect our conclusions in any way. As we expected, DIMs are comparable in performance to flooding when all sizes of queries are equally likely. For an algebraic distribution of query sizes, the relative performance is close to that for the exponential distribution. For normally distributed events, the insertion costs

[6]In practice, the performance of local replication is likely to be much better than this. Assuming a node and its replica don't simultaneously fail often, a node will almost always detect a replica failure and re-replicate, leading to near 100% response rates.

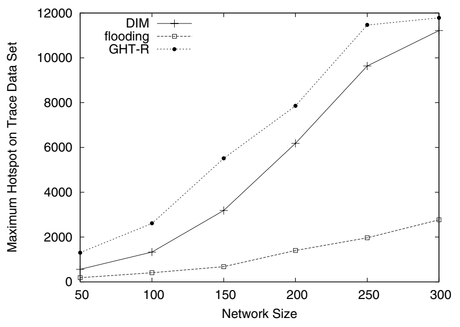

Figure 15: Hotspot usage

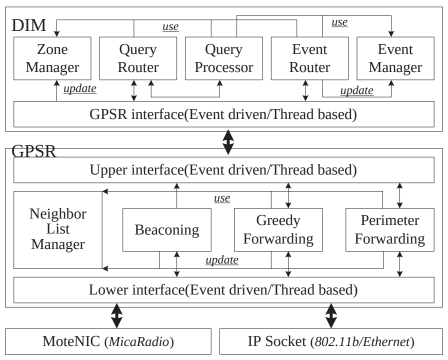

Figure 16: Software architecture of DIM over GPSR

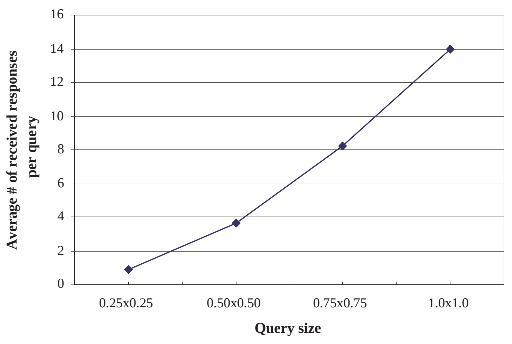

Figure 17: Number of events received for different query sizes

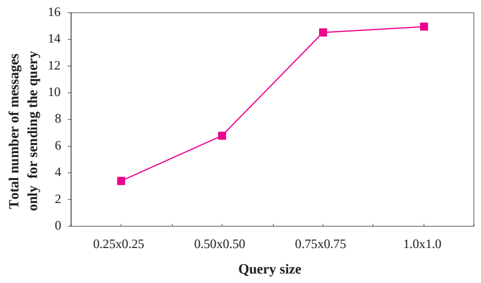

Figure 18: Query distribution cost

are comparable to that for the uniform distribution.

Finally, we note that in all our evaluations we have only used *list* queries (those that request all events matching the specified range). We expect that for *summary* queries (those that expect an aggregate over matching events), the overall cost of DIMs could be lower because the matching data are likely to be found in one or a small number of zones. We leave an understanding of this to future work. Also left to future work is a detailed understanding of the impact of location error on DIM's mechanisms. Recent work [22] has examined the impact of imprecise location information on other data-centric storage mechanisms such as GHTs, and found that there exist relatively simple fixes to GPSR that ameliorate the effects of location error.

6. IMPLEMENTATION

We have implemented DIMs on a Linux platform suitable for experimentation on PDAs and PC-104 class machines. To implement DIMs, we had to develop and test an independent implementation of GPSR. Our GPSR implementation is full-featured, while our DIM implementation has most of the algorithms discussed in Section 3; some of the robustness extensions have only simpler variants implemented.

The software architecture of DIM/GPSR system is shown in Figure 16. The entire system (about 5000 lines of code) is event-driven and multi-threaded. The DIM subsystem consists of six logical components: zone management, event maintenance, event routing, query routing, query processing, and GPSR interactions. The GPSR system is implemented as user-level daemon process. Applications are executed as clients. For the DIM subsystem, the GPSR module provides several extensions: it exports information about neighbors, and provides callbacks during packet forwarding and perimeter-mode termination.

We tested our implementation on a testbed consisting of 8 PC-104 class machines. Each of these boxes runs Linux and uses a Mica mote (attached through a serial cable) for communication. These boxes are laid out in an office building with a total spatial separation of over a hundred feet. We manually measured the locations of these nodes relative to some coordinate system and configured the nodes with their location. The network topology is approximately a chain.

On this testbed, we inserted queries and events from a single designated node. Our events have two attributes which span all combinations of the four values $[0, 0.25, 0.75, 1]$ (sixteen events in all). Our queries span four sizes, returning 1, 4, 9 and 16 events respectively.

Figure 17 plots the number of events received for different sized queries. It might appear that we received fewer events than expected, but this graph doesn't count the events that were already stored at the querier. With that adjustment, the number of responses matches our expectation. Finally, Figure 18 shows the total number of messages required for different query sizes on our testbed.

While these experiments do not reveal as much about the performance range of DIMs as our simulations do, they nevertheless serve as proof-of-concept for DIMs. Our next step in the implementation is to port DIMs to the Mica motes, and integrate them into the TinyDB [17] sensor database engine on motes.

7. CONCLUSIONS

In this paper, we have discussed the design and evaluation of a distributed data structure called DIM for efficiently resolving multi-dimensional range queries in sensor networks. Our design of DIMs relies upon a novel locality-preserving hash inspired by early work in database indexing, and is built upon GPSR. We have a working prototype, both of GPSR and DIM, and plan to conduct larger scale experiments in the future.

There are several interesting future directions that we intend to pursue. One is adaptation to skewed data distributions, since these can cause storage and transmission hotspots. Unlike traditional database indices that re-balance trees upon data insertion, in sensor networks it might be feasible to re-structure the zones on a much larger timescale after obtaining a rough global estimate of the data distribution. Another direction is support for node heterogeneity in the zone construction process; nodes with larger storage space assert larger-sized zones for themselves. A third is support for efficient resolution of *existential* queries—whether there exists an event matching a multi-dimensional range.

Acknowledgments

This work benefited greatly from discussions with Fang Bian, Hui Zhang and other ENL lab members, as well as from comments provided by the reviewers and our shepherd Feng Zhao.

8. REFERENCES

[1] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Baltimore, MD, January 2003.

[2] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communicaions of the ACM*, 18(9):475–484, 1975.

[3] P. Bonnet, J. E. Gerhke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability.* Springer, New York, 2001.

[5] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[6] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4:1–9, 1974.

[7] D. Ganesan, D. Estrin, and J. Heidemann. DIMENSIONS: Why do we need a new Data Handling architecture for Sensor Networks? In *Proceedings of the First Workshop on Hot Topics In Networks (HotNets-1)*, Princeton, NJ, October 2002.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th VLDB conference*, Edinburgh, Scotland, September 1999.

[9] R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker. The Sensor Network as a Database. Technical Report 02-771, Computer Science Department, University of Southern California, September 2002.

[10] B. Greenstein, D. Estrin, R. Govindan, S. Ratnasamy, and S. Shenker. DIFS: A Distributed Index for Features in Sensor Networks. In *Proceedings of 1st IEEE International Workshop on Sensor Network Protocols and Applications*, Anchorage, AK, May 2003.

[11] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD*, Boston, MA, June 1984.

[12] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, volume 2429 of *LNCS*, page 242, Cambridge, MA, March 2002. Springer-Verlag.

[13] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, Dallas, Texas, May 1998.

[14] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving Hashing in Multidimensional Spaces. In *Proceedings of the 29th Annual ACM symposium on Theory of Computing*, pages 618 – 625, El Paso, Texas, May 1997. ACM Press.

[15] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, Boston, MA, August 2000.

[16] B. Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 2000)*, Boston, MA, August 2000.

[17] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of ACM SIGCMOD*, San Diego, CA, June 2003.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks. In *Proceedings of 5th Annual Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[20] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A Geographic Hash Table for Data-Centric Storage. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[21] H. Samet. Spatial Data Structures. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 361–385. Addison Wesley/ACM, 1995.

[22] K. Sead, A. Helmy, and R. Govindan. On the Effect of Localization Errors on Geographic Face Routing in Sensor Networks. In *Under submission*, 2003.

[23] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-Centric Storage in Sensornets. In *Proc. ACM SIGCOMM Workshop on Hot Topics In Networks*, Princeton, NJ, 2002.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[25] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang. A Two-Tier Data Dissemination Model for Large-scale Wireless Sensor Networks. In *Proceedings of the Eighth Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom'02)*, Atlanta, GA, September 2002.