An Interrelational Grouping Abstraction for Heterogeneous Sensors

VASANTH RAJAMANI, SANEM KABADAYI, and CHRISTINE JULIEN
The University of Texas at Austin

In wireless sensor network applications, the potential to use cooperation to resolve user queries remains largely untapped. Efficiently answering a user's questions requires identifying the correct set of nodes that can answer the question and enabling coordination between them. In this article, we propose a *query domain* abstraction that allows an application to dynamically specify the nodes best suited to answering a particular query. Selecting the ideal set of heterogeneous sensors entails answering two fundamental questions—*how* are the selected sensors related to one another, and *where* should the resulting sensor coalition be located. We introduce two abstractions, the *proximity function* and the *reference function*, to precisely specify each of these concerns within a query. All nodes in the query domain must satisfy any provided proximity function, a user-defined function that constrains the relative relationship among the group of nodes (e.g., based on a property of the network or physical environment or on logical properties of the nodes). The selected set of nodes must also satisfy any provided reference function, a mechanism to scope the location of the query domain to a specified area of interest (e.g., within a certain distance from a specified reference point). In this article, we model these abstractions and present a set of protocols that accomplish this task with varying degrees of correctness. We evaluate their performance through simulation and highlight the tradeoffs between protocol overhead and correctness.

27

1. INTRODUCTION

Miniaturization has enabled the production of inexpensive battery-operated sensors that contain at least one module to sense various aspects of the physical environment, such as temperature and humidity. A sensor network consists of a connected set of sensors, each of which can perform one or more specialized sensing tasks, dynamically form a multihop network, perform some general-purpose computation, and store limited amounts of data. Sensor networks show great promise in monitoring the physical world in real time. When sensor networks are deployed on a large scale, however, there is an explosion in the amount of data to observe and analyze. Just as the plethora of Web data was largely human-unusable until the advent of modern search engines, querying techniques will play a pivotal role in comprehending sensor data.

Sensor networks have been deployed in a wide range of applications such as habitat monitoring [Mainwaring et al. 2002], intelligent construction sites [Hammer et al. 2006], and industrial sensing [Krishnamurthy et al. 2005]. A common theme that emerges when studying these applications is the need to retrieve temporally correlated but heterogeneous data to answer a particular question. Consider a scenario where sensors are deployed on a crane on a construction site to ensure that the crane has not exceeded its safe working capacity. Sensor data as diverse as wind speed, load weight, and boom angle need to be accumulated and processed to monitor safety [Neitzel et al. 2001]. The resulting aggregate information can give a crane operator an indication of a potentially dangerous condition, which prevents the crane from toppling and endangering workers on the site.

As wireless sensor network deployments increase in such environments, it becomes imperative that query processing techniques efficiently handle heterogeneous data types. In a predeployed sensor network, nodes containing several sensor data acquisition boards may be present, but the set of sensors that a particular application query requires may be spread across multiple nodes in the network. Therefore, multiple sensing devices must often cooperate in the resolution of a single query. Practically, the sensors selected to work together should satisfy some application-specific constraints. In the crane safety example, the selected sensors need to be from the same crane, which is especially important when there are several cranes located close to one another. Such relationships between sensors that contain the desired sensing modules cannot be known a priori and are unique for each application. In addition, it is sometimes beneficial to specify where the cooperating group should be located relative to the client issuing the query or some other reference point. For example, in the crane example, the construction site manager may be interested in sensors that are all in a particular crane or sensors on a crane that is next to a densely populated building.

Many queries cannot be satisfactorily answered by existing query systems. TinyDB [Madden et al. 2005] is an example of a popular query processing system for sensor networks. Its traditional "HAVING" and "GROUP-BY" clauses allow grouping based on simple conjunctions over arithmetic comparison operators (e.g., "HAVING node-id $>$ 5"), but it is not possible to perform logical

grouping of nodes based on relative properties (e.g., forming a coalition of sensors where the distance between any two sensors is less than 10 meters). Other querying systems employ data centric routing (e.g., Directed Diffusion [Intanagonwiwat et al. 2000]) to satisfy requests that can be answered with homogeneous data types (e.g., the average temperature from a certain region in the network). Since a vast number of sensor network applications require the ability to deal with heterogeneous data types, these systems must be augmented with the ability to form logical grouping based on application properties.

In this article, we introduce an abstraction called the *query domain* that allows an application developer to logically group a subset of sensor nodes in the network. To help the user precisely define such a subset, we introduce two abstractions—the *proximity function* and the *reference function*. The proximity function specifies how nodes in the chosen subset are related to one another. It accomplishes this by allowing an application to specify the constraints between the sensors selected to answer a single query. These constraints are injected into the network, and sensor nodes that can satisfy the query's data type requirements *and* the proximity function's relationship constraints self-organize into a logical query domain. Specifying the constraints can potentially create several query domains in the sensor network. Often, the application developer is interested in choosing a particular query domain of interest. To allow him to constrain where the query domain can be located, we introduce the reference function. Reference functions can be based on the notion of geographic displacement, such as the distance from a particular object, or logical properties, such as association with a physical object.

One benefit of using dedicated query domains to answer queries is that we enable the same sensor network to be used for multiple applications. This diverges from current deployments where a sensor network is tailored for a particular application. Another benefit is that only the small subset of sensors in the query domain participate in replying to the query. Once organized, the query domain can be used as a handle to repeatedly ask the same question of the same coalition of sensors. This eliminates the communication overhead associated with creating new routing paths each time the query is issued, thus minimizing energy consumption. We demonstrate this by extending our work to handle *persistent queries* where a query domain, once created, provides periodic response to the query issuer.

The novel contributions of this work are as follows: First, we formally define three new abstractions, the *query domain*, the *proximity function*, and the *reference function*, that allow an application developer to identify the dynamic set of sensors that answer a query. Second, we devise a fully reactive approach that calculates and constructs the query domain completely on demand, enabling the networked sensors to self-organize. This protocol provides a practical realization of our abstractions. Third, we evaluate the effectiveness of our approach through simulation. Fourth, we investigate the correctness limitations of our proposed protocol and provide a variant that has stronger correctness guarantees, albeit at an increased overhead.

The remainder of the article is organized as follows. Section 2 motivates the problem with examples and provides a concrete problem description. Section 3

introduces our abstraction, and Section 4 presents its implementation. Section 5 discusses some instances where our implementation diverges from the formalization. Section 6 evaluates the performance of the protocol. Section 7 discusses related work, and Section 8 concludes.

2. MOTIVATION AND PROBLEM DEFINITION

We posit that the ability to collaboratively resolve a query is essential for effective query resolution. In this section, we provide concrete application examples where the stated issues materialize.

Crop Failure Prevention. Frequent crop failures are a cause of large monetary losses in modern farming. Most crop failures can be explained by variations in soil and climate [Mendelsohn 2007]. Monitoring and correlating temperature and soil conditions provides one way to detect the onset of crop failure. To perform such a task one needs heterogeneous devices such as temperature and capacitance sensors.[1] While it is possible that both temperature and capacitance sensors are available on the same device, it is also likely that the network contains devices with more limited capabilities. In such situations, a coalition of sensors can provide the same functionality as a single more complex sensor. The query for detecting deteriorating farming conditions may be stated as: "Select temperature and capacitance readings from sensors that are within 10 meters of one another." In this example, the user wishes to query temperature and capacitance sensors, and the constraint he designates is that the two sensors be near (within 10m of) one another. Note that this does not preclude the case that the two sensors are, in fact, located on the same device. In a similar fashion, some vineyards are currently experimenting with sensor networks to monitor grape crops [Millman 2004]. It may also be useful to specify where the sensor coalition should be located for it to be useful. The conditions to be monitored for each crop may depend on the nature of the crop. In defining the sensors to monitor, the farmer may want to specify that the sensor reading comes from a particular field of his farm or within a defined range of his current location.

Habitat Monitoring. Wireless sensor networks have been deployed in a variety of habitat monitoring environments [Mainwaring et al. 2002]. A field biologist may be interested in studying the mating behavior of species using a combination of audio and video sensors deployed in the field. The arrival of the animals may be detected by the audio sensors, which in turn trigger the video sensors to record and transmit images. It is critical that the latency between the audio and video sensors be within a stated threshold; a high latency can result in the reception of video images after the animal has left the area, leading to wasted video feeds. Also, the distance between the sensors must be small, or the audio sensors will detect the arrival of the animal, while the video sensors will record and transmit images from a completely different location. A query that accomplishes this task may take the form, "Select audio and video sensors such that they are within 20 feet of one another and the latency between the sensors is less than 1 second." In this case, knowledge about the

---

[1]http://www.soilsensor.com.

animals' habits may also allow such a query to be restricted to a given region of the environment where the animals are known to mate. This allows the distributed query processing to consume fewer network resources, boosting energy efficiency.

Intelligent Construction Site Safety.   Finally, sensor networks can be deployed to monitor safety conditions on a construction site [Neitzel et al. 2001]. As one example, sensors on a crane can monitor its load and warn the site manager if a crane is in danger of toppling. In addition to wind speed measurements, one needs to aggregate data such as the angle of the crane's arm, the position of the boom on the arm, and the weight of the load on the crane. In such a scenario, a useful query can be of the form, "Select all sensor readings from the same crane." While disparate sensors must be queried to monitor safety violations, the query domain construction must ensure that all readings are from the same crane and not from sensors positioned on distinct cranes that happen to be close to one another. In addition, the site manager may particularly be interested in the safety conditions of a crane next to a particular building, or a crane with a specific identification number.

This article addresses the problem of dynamically defining a sensor coalition that acts as a single entity in responding to an application's query. Since this must be done without any a priori knowledge, nodes must self-organize according to some application-specified rules provided at runtime. These rules help specify how the sensors are related to one another and to a given reference point.

3. ABSTRACTION

In this article, we propose the *query domain* abstraction to solve the problem we have outlined. In contrast to existing approaches that group nodes based solely on their content, our approach also groups nodes based on their relationships to each other and one or more reference points. The query domain defines a relationship that all responders to the query must satisfy among themselves. Distributing the definition of this relationship to nodes in a sensor network allows an application to dynamically impose an overlay structure on the network, impacting how the network behaves in response to the application's queries.

Figure 1 gives a pictorial depiction of a query domain. The members of the overlay (the gray area) are constrained by the application-specified *proximity function*. The query domain in the figure also contains a *reference function* that requires the resulting query domain (in this example) to be within 100m of the reference point in the picture. These constraints that define the query domain are attached to the query to allow for distributed computation. Because a query carries with it the definition of its query domain, nodes in the sensor network are not required to know a priori what types of query domains the application will request, alleviating the need to tailor a sensor network deployment to a particular application. Once set up, the query does not have to deal with individual nodes in the network. Instead, it can treat the entire overlay as a single node that can be queried repeatedly. In the remainder of this section,
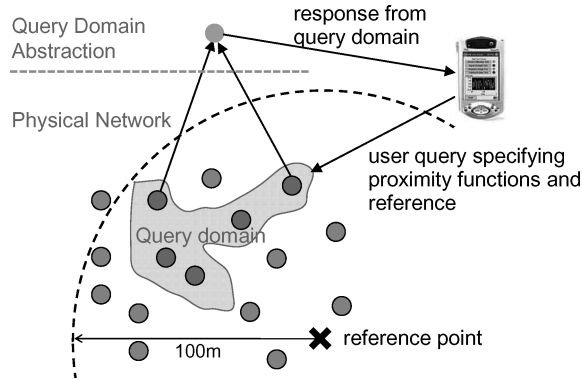
Fig. 1.   An example query domain.

we first formalize the query domain and its use of the proximity and reference functions. We then show how our application examples define query domains using these abstractions.

3.1 Formalization

Our abstraction can be formally specified using three components. The first component states the form of an application's query while the remaining two components formalize the semantics of the query domain supporting this query. Section 4 describes the protocols we use to implement this abstraction. Any query is defined first by the data types required (i.e., $t_1, t_1, \ldots, t_n$), second by the proximity function, F_p, applied to constrain the query domain, and lastly, by a reference function, F_r, used to provide a frame of reference for the location of the query domain. A query can be written as:

$$Q \; : \; \{t_1, t_2, \ldots, t_n\}/(F_p \vee F_r)^+.$$

F_p is an application-specified Boolean function that takes any pair of nodes and returns true if the pair satisfies the proximity function and false otherwise. Similarly, F_r is an application-specified Boolean function that takes one or more nodes and a reference point and returns true if the given nodes satisfy the proximity function and false otherwise. In addition to specifying the types, a valid query must specify at least one proximity function or reference function. A reference function can be of one of two types: an existential function or a complete function. For the former, any member of the query domain must satisfy the reference function; for the latter, every member of the query domain must satisfy the reference function. When referring to components of a query, we use the "dot" notation. That is, the set of types a query requires $(t_1, t_2, \ldots, t_n)$ can be retrieved using Q.types, while the proximity function and reference functions can be retrieved using $Q.F_p$ and $Q.F_r$ respectively. The set of types any one sensor in the network can provide is referenced in a similar way, as s.types.

Formally, a query domain for a query, Q, is any set, S, of nodes that satisfies the following four conditions:[2]

$$\langle \forall s : s \in S :: s.\text{types} \cap Q.\text{types} \neq \emptyset \rangle \wedge$$
$$\langle \forall t : t \in Q.\text{types} :: \langle \exists s \in S :: t \in s.\text{types} \rangle \rangle \wedge$$
$$\langle \forall s : s \in S :: |(S - \{s\}).\text{types} \cap Q.\text{types}| < |Q.\text{types}| \rangle \wedge$$
$$\langle \forall s_1, s_2 : s_1, s_2 \in S :: Q.F_p(s_1, s_2) \rangle.$$

The first of these constraints requires that each sensor in the query domain S have at least one of the query's required types. The second condition requires every type requested in Q to be provided by at least one sensor in S. The third constraint ensures that none of the sensors selected for the query domain are redundant. That is, this condition guarantees that if any sensor s is removed from S, then the types remaining in S are insufficient to satisfy Q. (In this condition, the notation $|S|$ refers to the cardinality of the set S.) The fourth condition ensures that the proximity function F_p is true for every pair of sensors. This also implicitly requires that F_p be reflexive, that is, $F_p(s, s)$ is true.

The above conditions are sufficient if no reference function is provided. However, if a reference function is specified, an additional condition must also be specified. The nature of this condition depends on which of the two types of reference function is applied. For existential reference functions, the following must be true:

$$\langle \exists s : s \in S :: Q.F_r(s) \rangle.$$

Alternatively, a complete proximity function requires the following constraint:

$$\langle \forall s : s \in S :: Q.F_r(s) \rangle.$$

The reference function also provides a convenient lever to scope the search space over which a query executes. It can be used to express a network time to live (TTL) that is commonly used to scope protocols:

$$\text{hopCountReference}_{\forall, client, maxHops}.$$

A reference function of this form ensures that every node in the query domain is within *maxHops* of the client issuing the query. Such a reference function can be easily tagged to all queries so as to limit the overhead associated with query execution. This is especially useful, when the proximity function is not a function of network properties but relies instead on logical properties.

These definitions provide the necessary and sufficient conditions for membership in a query domain. The definition does not necessarily reflect the ease or efficiency with which a particular group of nodes can be connected from a network perspective. We revisit this in greater detail in Section 5.

---

[2]In the three-part notation: $\langle$op *quantified_variables* : *range* :: *expression*$\rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which op is applied. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for op, for example, *true* if op is $\forall$.

3.2 Application Examples

Given this formalization of queries, we next show how the applications described in Section 2 can use our model to easily specify their query requirements.

Crop Failure Prevention. Detecting and preventing crop failure requires monitoring temperature and capacitance readings from sensors that are located physically close together. This query has the form:

$$Q \; : \; \{\text{temperature, capacitance}\}/\text{distanceProximity}_{10m},$$

where the types within the braces are the two data types the application must find in the network, while distanceProximity refers to a (built-in) proximity function that requires all of the sensors in the query domain to be within a given distance of each other. For proximity functions that require such thresholds, the threshold (e.g., 10 meters) is given as a subscript. Formally, this indicates an infinite number of distanceProximity functions (one for each possible value of the threshold). This is implemented as a parameterized function. It is also possible that a proximity function requires more than one threshold; in such a case, the subscripts are separated by commas and refer to multiple parameters.

In some applications it may be necessary to specify where the query domain should be constructed. For example, the farmer may wish to detect crop failure only within a particular field as defined as a range around a given location. To accomplish this, we augment the query above with a reference function:

$$Q \; : \; \{\{\text{temperature, humidity}\}/\text{distanceProximity}_{10m}\}$$
$$/\text{distanceReference}_{\forall,(30,50),100}.$$

Similar to distanceProximity, the distanceReference is a built-in function that allows the user to specify that the query domain should be within a certain distance of a location of interest. In this example, the threshold (100) constrains the query domain to be within 100m of the location coordinates (30,50) (a parameter). The subscript $\forall$ designates the complete reference function, forcing the query domain to ensure that every sensor in the query domain is within 100m of the coordinates (30,50).

Habitat Monitoring. For the habitat monitoring application example, we require audio and video sensors that are connected by network paths that provide some maximum latency (for example, one second):

$$Q \; : \; \{\text{video, audio}\}/\text{latencyProximity}_{1s}.$$

Applications may also require that the sensors in the query domain be physically close together to provide some confidence that the video sensor is capturing relevant video. To augment the above to account for distance in addition to latency, we simply apply a second cost function:

$$Q \; : \; \{\{\text{video, audio}\}/\text{latencyProximity}_{1s}\}$$
$$/\text{distanceProximity}_{10m}.$$

Logically, these two proximity functions are combined into one larger function with the cumulative constraints of each function. Alternatively, the

fourth condition in our query domain formalization could be rewritten as:

$$\langle \forall s_1, s_2, i : s_1, s_2 \in S \wedge i < m :: Q.F_{p_i}(s_1, s_2) \rangle,$$

where m refers to the number of proximity functions applied to the query.

The query can once again be augmented with a reference function to specify where the desired query domain should reside in the network.

Intelligent Construction Site Safety. For the crane safety application example, the query monitors crane data to ensure that safety conditions on the site are satisfied. To do this, the application must collect information from a variety of sensors on the crane, for example, the wind speed, the boom angle, and load weight [Hammer et al. 2006]:

$$Q \; : \; \{\text{speed, boom, weight}\}/\text{contextProximity}_{\text{SameCrane}}.$$

The proximity function applied in this case is significantly different from the proximity functions used in the previous two examples. The contextProximity function used here designates a logical relationship between the sensors selected as opposed to a physical constraint on the network or environment. In this case, contextProximity$_{\text{SameCrane}}$ ensures that all sensors selected for the query domain are from the same crane. While query domains, in general, do not *require* explicit tagging, in this case, the application expects each sensor on a crane to be tagged with the identity of the crane in question.

This query will form a query domain where all the required sensors reside on the same crane. More often than not, the user is interested in the safety consideration of a particular crane. For example, the querier might be interested in a query domain where all the sensors reside on the crane with identification number six. In such a situation the query can be simply represented as follows:

$$Q \; : \; \{\text{speed, boom, weight}\}/\text{contextReference}_{\forall,\text{CraneSix}}.$$

We are able to specify the query domain without a proximity function because the reference function makes the proximity function redundant. If all the sensors are on crane six, clearly they reside on the same crane.

3.3 Programming Query Domains

We now demonstrate how a user would employ our abstractions to specify query domains. Listing API 1 demonstrates this API using the Java language. The **DataListener** interface used in the query domain API is the mechanism through which the query domain returns its members' data values. The user can call the **issueQuery** and **issuePersistentQuery** methods to instantiate one-shot and persistent[3] queries respectively. Listing Program 1 shows how a programmer could use this API to create query domains from our application examples.[4] In the example code, we have abbreviated **QueryDomain** to **QD**, **ProximityFunction** to **PF** and **ReferenceFunction** to **RF** respectively.

---

[3]Persistent queries are discussed in detail in Section 6.4

[4]These examples rely on the assumption of predefined proximity and reference functions available as library components. The more sophisticated Java to nesC translator necessary for translation of arbitrarily defined functions is outside the scope of this work.

Table I. Definitions

Query's Type Set (Q.types)	The set of sensor types required for this query
Proximity Function ($Q. F_p$)	The user-defined constraint on membership in the query domain
Reference Function ($Q. F_r$)	The user-defined constraint on location of the query domain
Path (P_{ij})	A series of links in the network that transitively connects nodes i and j
Return Path (P)	A connected graph of nodes selected for a query domain

---

API 1 Query Domain API

```
class QueryDomain {
    //Constructor
    public QueryDomain(String[] dataTypes, ProximityFunction[] pf,
                       ReferenceFunction[] rf);
    //Methods
    public void issueQuery(DataListener response);
    public void issuePersistentQuery(int freq,DataListener response);
  }
}
```

---

4. PROTOCOL

In this section, we present a protocol that uses our abstractions to dynamically create query domains. Table I presents a summary of the abbreviations we will use in describing this protocol, some of which were introduced in Section 3.

The goal of a protocol for forming a query domain is to locate a set of nodes, S, that can satisfy a query specified as described in the previous section. The protocol should return to the query issuer a return path, P, that the querier can subsequently use to contact the constructed query domain. This path is not necessary in every case, but if the application intends to use the same query domain again, having a preconstructed path that it can use to contact the query domain in the future can significantly decrease the overhead and latency. The benefits of this approach will be evaluated in Section 6.

Our completely reactive protocol for query domain construction sends an initial flood looking for a sensor that can satisfy at least one of the data type requirements (i.e., a sensor node that supports any one of the types in Q.types). If a reference function is provided, the protocol verifies that the first node satisfies the reference function. After the first member of the query domain is found, this member itself initiates a secondary flood looking for the remaining unsatisfied types from Q.types. The second flood is explicitly constrained by the proximity function, F_p. It is also constrained by the reference function F_r if the specification of the reference function used the rule stipulating that all the nodes in the query domain satisfy the reference function ($\forall$ rule).

Program 1 Example Code

```
// Crop Failure with no reference function
//Uses the built in distance proximity
//such that the total distance so far is less than 100
PF farmingPF = new PF(PF.PF_DISTANCE, PF.DFORMULA,
                      new IntegerThreshold(100));
QD farmingQD = new QD({temperature, capacitance},{farmingPF}, NULL);
farmingQD.issueQuery(answerListener);

//Same query with an added reference function
//all nodes have to be within 100m of (30,50)
RF farmingRF = new RF(RF.RF_DISTANCE,RF.RF_ALL,new IntegerCoordX(30),
                      new IntegerCoordY(50), RF.RF_DFORMULA,
                      new IntegerThreshold(100));
QD farmingQD = new QD({temperature, capacitance},
                      {farmingPF}, {farmingRF});
farmingQD.issueQuery(answerListener);
-------------------------------------------------------------------
//Habitat monitoring query with two proximity functions
// latency has to be less than one second
//i.e., sum of latency on a path cannot exceed threshold
PF habitatPF1 = new PF(PF.PF_LATENCY, PF.PF_SUM,
                       new IntegerThreshold(1));
// distance has to be less than 10m
PF habitatPF2 = new PF(PF.PF_DISTANCE, PF.DFORMULA,
                       new IntegerThreshold(10));
QD habitatQD = new QD({audio, video},{habitatPF1, habitatPF2}, NULL);
habitatQD.issueQuery(answerListener);
-------------------------------------------------------------------
//Intelligent Construction example using just a reference function
RF constructionRF = new RF(RF.RF_CONTEXT, RF.SAMETAG,new IntegerTag(6));
QD constructionQD = new QD({speed, boom, weight}, NULL,
                              {constructionRF});
//issuing a persistent query with frequency one query every minute
constructionQD.issuePersistentQuery(60, answerListener);
```

As these secondary floods are initiated, the query keeps track of the path it has traversed so far. When all of the types from Q.types have been satisfied, the final sensor in the chain sends the return path P to the query initiator. This return message contains the values for each of the sensors in the query domain *and* the information needed for the query issuer to issue subsequent queries to the same domain. As an example, the query in Figure 2 receives two satisfactory query domains. A third node matches one part of the query but fails to match the remainder, so it does not return a query domain. We note that there may be multiple (possibly overlapping) query domains that satisfy a single query. To specify any particular one of interest, the user can leverage the reference function to constrain the query. In our protocols, the query issuer selects the first response it receives (favoring query domains with low latency) that satisfy both the proximity and reference functions. In resource-constrained sensor networks, it is impractical to remember all other responses. However,
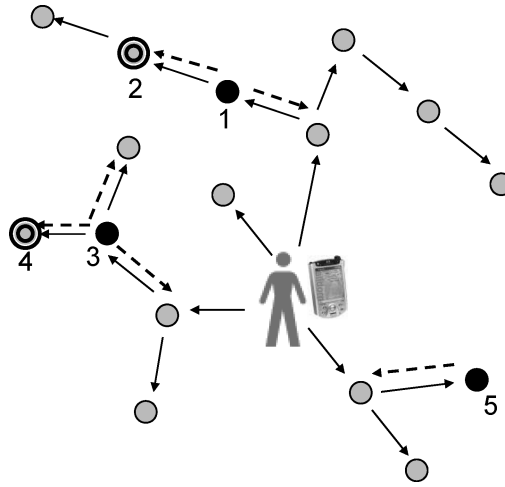
Fig. 2. High-level protocol. Solid arrows indicate the initial flood, while dashed arrows indicate the secondary floods (which, for simplicity, were limited to a one-hop radius). Black nodes indicate the first node in a query domain to match one of the data types; nodes with the double boundary matched the second data type. Nodes 1 and 2 form one query domain, while nodes 3 and 4 form another. Node 5 satisfied one of the requested data types, but could not find the other data type within one hop.
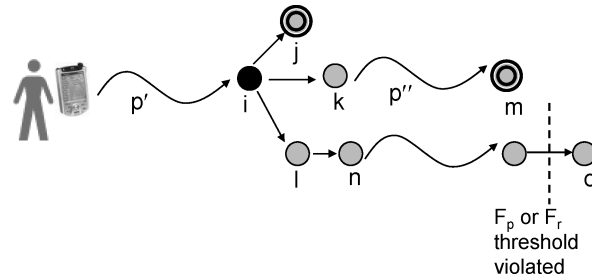

Fig. 3. The three cases a query domain resolution can encounter.

the protocol outlined can be easily augmented to remember a few responses in case the selected query domain fails in subsequent queries.

Figure 3 shows this process, focusing on three cases a query domain resolution may encounter. In this analysis, we consider a query that requests only two data types; queries for more types of data incur subsequent constrained floods. The first case (the path involving node j) is our base case. In this case, node i was a hit for one of the data types in Q.types. After this match, node i initiated a secondary query. Node j matches the second data type required by Q.types, and j sends the return path $j - i - p'$ back to the query issuer, where p' is the series of hops the query had to traverse before reaching node i.

For the second case, consider the path shown to node m. In this case, node i initially matched one of the data types of the query. Again, i initiated a secondary query, but k (and all the nodes between k and m) could not satisfy the remainder of the query. These nodes continued to forward the query until it

landed at node m, which could provide the second data type. Node m returns to the query issuer the return path $m - p'' - k - i - p'$, giving the query issuer access to the entire query domain, the set $\{i, m\}$. The return path itself is not the query domain, but it enables the query issuer to contact every device that is within the query domain.

In the final case, the path terminating at the node labeled o, node i again satisfied one of the two required data types for the query. However, in forwarding the query in search of the second required data type, either the proximity function or the reference function was violated along the path. In this case, no return path is sent back to the query issuer because no query domain was formed along this search route.

4.1 Worst Case Overhead

The overhead of the protocol depends on the number of types, and the nature of the proximity and reference functions. In the worst case, our protocol can trigger a new flood for every type requested in the query resulting in $O(n^k)$ messages in the network, where n is the number of nodes and k is the number of types requested. If the proximity function is a network property (e.g., hop-count) or environmental property (e.g., distance), the proximity function naturally scopes the successive floods ensuring that they are not network wide. In the case of logical properties (e.g., sameCrane), the proximity function does not serve the dual purpose of attesting membership and scoping the floods. However, as shown in the previous section, the reference function can be used as an effective scoping mechanism to restrict excessive flooding. Even if the user employs a simple reference function that serves as a network TTL, the worst case overhead of the protocol can be changed dramatically. It can be reduced to $O(m^k)$, where m is the number of nodes allowed in the scoped region. The overhead associated with the protocol can be reduced substantially by scoping the protocol in a way such that $m \ll n$.

4.2 Query Domain Failure

In some instances, it is possible for our protocol to fail to locate a query domain even if a valid one exists. This is due to the fact that our protocol's behavior is conservative. We discuss and evaluate this conservativeness in detail in Sections 5 and 6.5, respectively.

Regardless of the cause, there are a few plausible alternatives that a user can take in such a scenario.

—The user can do nothing.
—The user can reissue the query with relaxed constraints on the proximity or reference functions.
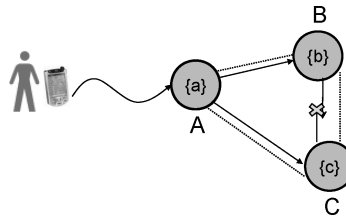
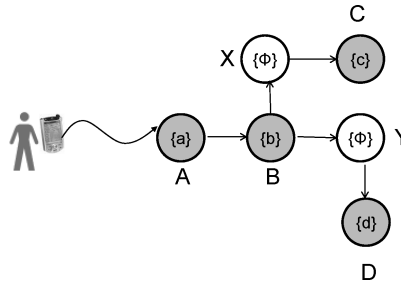Fig. 4. Conservativeness on a nonbranching path.

Fig. 5. Conservativeness when the query domain is not constrained to a single nonbranching path.

—If the query is critical enough to take a more expensive course of action, the user can flood the network, retrieve all values, and perform a centralized grouping at a powerful base station node.

In our implementation, the query framework accepts a parameter in the form of a Boolean flag that indicates whether a query domain failure should result in protracted flooding and centralized grouping. This flag can be set by the user depending on how much value he places on successfully forming a valid query domain.

The complete overhead associated with our protocol is a function of the scoping mechanism used and the action undertaken when the protocol fails to detect a valid query domain. In Section 6, we evaluate our protocol by employing the most expensive fall back option and varying the scoping mechanism.

5. CONSERVATIVENESS OF QUERY DOMAINS

As we will demonstrate in the next section, the protocol in Section 4 is an efficient protocol for finding and returning a large proportion of the available query domains. However, it is more conservative than the formalization in Section 3 in that it does not guarantee that it finds *every* satisfactory query domain. In this section, we discuss limitations resulting from this design choice. In Section 6.5, we quantify these limitations and show that our protocol has significant benefits that outweigh these drawbacks.

One alternative to our protocol is to use a more sophisticated centralized protocol. A complex centralized protocol can occasionally find more legal query domains that our protocol overlooks. We use Figure 4 and Figure 5 to examine this further.
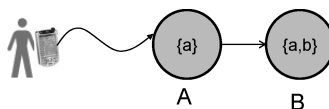
Fig. 6. Conservativeness due to absence of redundancy removal.

In Figure 4, node **A** can provide sensor a, node **B** can provide sensor b, and node **C** can provide sensor c. The dashed lines indicate connectivity between the nodes. Consider a query where a user is interested in all three of these data types and requires that all members of the query domain be within a single hop of each other. The set of nodes {**A**, **B**, **C**} does, in fact, satisfy the proximity function as defined by our formalization. The nodes are all within one hop of one another, and collectively, they contain all of the desired sensor types. In the figure, the distributed protocol first detects sensor a at node **A**. When node A initiates the second constrained flood, it finds sensor b at node **B** and sensor c at node **C** along two different paths. However, nodes **B** and **C** will not rebroadcast the message to each other. The pairs **A**-**B** and **A**-**C** are each exactly one hop apart, and any further broadcast to nodes **C** or **B**, respectively, will violate the proximity function along the path through **A**. In this example, our protocol is unable to detect nodes {**A**, **B**, **C**} as a valid query domain.

To generalize the statement of this conservativeness, we say that our protocol is more conservative than the formalization in Section 3 when the query domain is not constrained to a single nonbranching path. Figure 5 shows another example. Nodes **A**, **B**, **C** and **D** contain sensors a, b, c, and d respectively. Nodes **X** and **Y** do not contain any of the desired sensors, but are in the connectivity path. The user queries for sensors a, b, c and d with a proximity function that requires all nodes to be within four hops of one another. The set of nodes {**A**, **B**, **C**, **D**} form a perfectly valid query domain. In the figure, our protocol will first find sensor a at node **A** and then sensor b at node **B**. Subsequently, there will be a fork and the next node will be either **X** or **Y**. Since there can be one more node in the return path without violating the proximity function, the next node to be added to the return path is either node **C** or node **D**. While **C** and **D** each contain one of the desired types, neither {**A**, **B**, **X**, **C**} nor {**A**, **B**, **Y**, **D**} will be a valid query domain as neither contains all four desired sensor types. In this example, our protocol will not detect a query domain because the query domain topography does not conform to a path-based structure.

In Section 6.5, we evaluate how often query domains result in topological formations like trees or directed acyclic graphs instead of single network paths. We also implement a protocol which makes use of a centralized coordinator that can identify query domains mentioned in the examples above and compare it to our reactive path-based protocol.

Another way our protocol is a more conservative than formalized is due to the fact that sensors selected to form a query domain may be redundant. Our protocol provides no facility for removing sensors if their capabilities prove to overlap those of other sensors that must be in the query domain. We use Figure 6 as a reference to demonstrate this point. Consider a query that requires only two types: {a, b}. If the query issuer discovers sensor a at node **A**, the protocol

will start looking for sensor b. Since our implementation removes a from the required sensor list, the protocol will henceforth look for sensor b only. When the protocol discovers a node (e.g., node **B**), that can provide the desired type b, the resulting query domain is {**A**, **B**}. Node **A** is clearly redundant in this case, since node **B** contains every sensor provided by node **A**. However, our protocol does not remove the redundant node (node **A**) from the query domain returned to the user. This is largely an implementation detail and can be easily overcome. However, doing so would entail sending more information in every packet, increasing the protocol overhead. The added complexity adds very little value in practice, as the redundant node is still likely to remain in the return path to the user. Therefore, neither the overhead nor the latency is reduced. As the added complexity does not result in any savings, we do not investigate protocols to address this issue.

6. EVALUATION

In this section, we provide an evaluation of our protocol. Our protocol's code size is 16,980 bytes in ROM and 2,568 bytes in RAM memory. To thoroughly evaluate our protocol, we used the OMNeT++ network simulator[5] and its mobility framework.[6] We employed the NesCT language translator[7] to translate TinyOS [Hill et al. 2000] code written for MICA2 motes[8] into C++ code that can be run on OMNeT++ simulator. We have also tested our code using the TOSSIM network simulator [Levis et al. 2003], which allows direct simulation of TinyOS [Hill et al. 2000]. The TOSSIM network simulator uses a CSMA MAC protocol and allows for bit level simulation. OMNeT++ uses a 802.11 MAC layer that models packet level transmissions and allows for better testing with probabilistic faults. The results presented here use the NesCT tool as it allows for more robust testing of dense sensor network deployments. The results presented here include any additional overhead associated with using these simulators, like the use of RTS/CTS messages, in the case of the OMNeT++ simulator.

6.1 Simulation Setup

We used a uniform random placement of sensor nodes in a 100 foot $\times$ 100 foot area. We used the OMNeT++ radio model that accounts for packet collisions and fading. To distribute sensor types, we took all of the possible types for a particular simulation, created all possible combinations of those types, and randomly dispersed the combinations in the network. For example, when there are three possible sensor types, there are seven possible combinations of sensors (a node can have one of any three of the types, two of any three of the types, or all three types). Consider a network where the possible sensor types are {a, b, c}. In a simulation layout of 50 nodes, seven nodes would provide only

---

data type a, seven would provide only type b, seven would provide only type c, seven would provide types a and b, etc. The last, leftover node randomly provides one of the seven combinations.

6.2 Performance Metrics

To evaluate our protocol, we use three performance metrics: (i) the number of messages sent per query, (ii) the query response time, and (iii) and the percentage of queries successfully resolved. Since wireless sensor networks are extremely resource-constrained, it is essential for a protocol to minimize message overhead to save energy. In addition, the usefulness of the protocol is at least partially dependent on the latency associated with retrieving the results.

We have evaluated the power of our abstractions through several categories of tests. The first set of simulations measures the overhead and latency associated with our protocol. In the second set, we evaluate how our protocol performs in resolving *persistent* queries. Finally, we compare our protocol against a more complex centralized protocol and evaluate the conservativeness issues raised in Section 5. The error bars on the graphs indicate 95% confidence intervals.

6.3 Basic Protocol Performance

In these simulations, we compare our protocol to another protocol using the performance metrics outlined above.

6.3.1 *Choice of Comparison.* In evaluating our protocol's capabilities and performance trade-offs, we selected a flooding-based protocol as a benchmark. This protocol broadcasts the required sensor types to every node within communication range of the query issuer. Recipients of the broadcast reply if they have all the required sensors. Otherwise, the message is rebroadcast until the required sensors are found or the network's TTL is reached. Flooding provides a good benchmark because existing implementations of the applications described in this paper typically use this technique to resolve their queries. However, simple flooding is too inefficient from a message overhead perspective. Therefore, we compare our protocol against an intelligent flooding protocol that accomplishes the same task with much lower overhead [Liang et al. 2007]. This protocol uses far fewer forwarding nodes than a simple, naïve flooding implementation.

6.3.2 *Comparison Results.* To evaluate our performance metrics, we varied the number of nodes from 50 to 500. Table II shows that this results in the average number of neighbors per node changing from approximately one to fifteen in our simulation setup, thus demonstrating that we vary the network density from sparse to dense in different experiments. To study the feasibility of creating query domains using proximity functions, we implemented two types of query domains. While the first uses a simple proximity function that requires that all nodes be within three hops of one another, the majority of our reported results use the second, more sophisticated query domain where the proximity function specifies that the distance between each member is no greater than
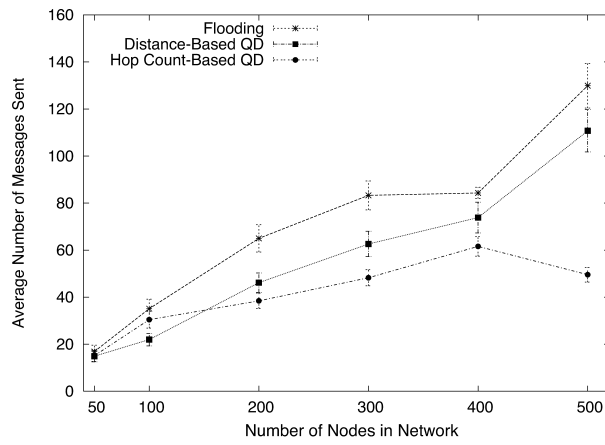
Table II. Relationship Between Node
Density and Number of Neighbors per
Node

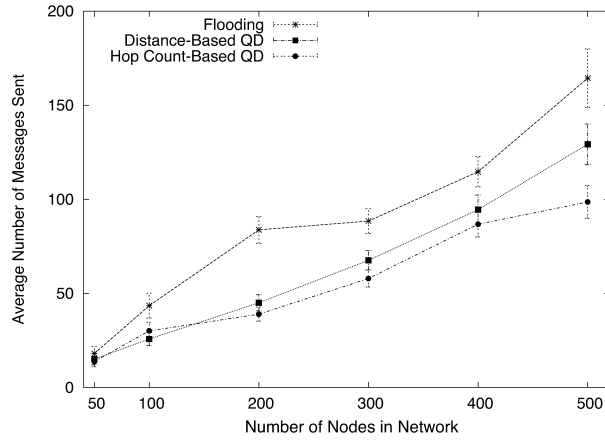| Nodes	Number of Neighbors
50	0.98
100	2.16
200	5.23
300	8.26
400	11.41
500	15.34

forty feet. In conjunction with these proximity functions, we use a simple reference function that serves as a network TTL (as described in Section 3). It forces all the nodes in the query domain to be a within a certain number of hops from the query issuer. We varied the number of hops allowed by the reference function from three to seven. We tested our protocol with queries of different complexities by varying the number of sensors required in the query. The results presented in Figures 7 and 8 are for a query that requires four sensor types and employs the proximity and reference functions mentioned above.

Figure 7 shows the number of messages transmitted by all nodes in the network to resolve a query for our protocol and for the intelligent flooding protocol described previously. The intelligent flooding algorithm requires proactive exchange of beacon messages to determine the optimal set of nodes that should continue to propagate the broadcast message. Our protocol is entirely reactive and does not require such beacon messages. Figure 7 does not include the overhead associated with beacon exchange in the intelligent flooding protocol because we view this as a one time setup cost and not the cost for operation. In the graphs shown in Figure 7, both protocols observed the same behavior if they failed to locate a valid query domain using the standard discovery protocol; in response to failure, both protocols repeatedly flooded the network for individual sensors and performed a centralized grouping.
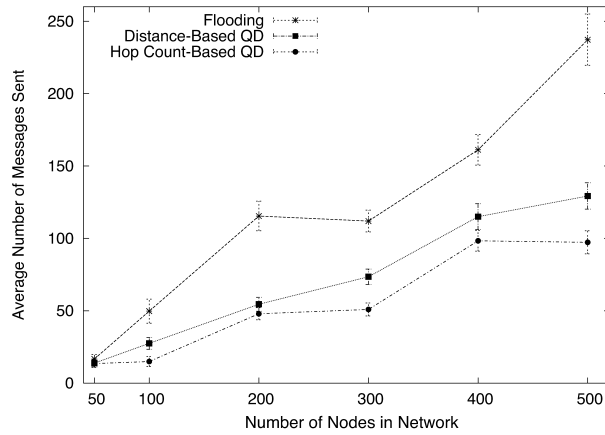
In Figure 7, the number of messages increases as the number of nodes increases because with increasing density, the number of nodes in rebroadcasting range increases. Regardless of the particular proximity or reference function used, our protocol outperforms the flooding based protocol, especially as the number of nodes in the network increases because the overhead is greatly affected by how often each protocol succeeds in resolving the query successfully. Our protocol accomplishes this task far more frequently than the alternative flooding based protocol. As a result, the number of times the protocol has to execute the expensive secondary protocol that results in repeated flooding of the network is drastically reduced. This is especially true for networks with lower node densities or when the TTL is set to a low value because our protocol does not need to search as widely for a match as nodes cooperate in resolving the query. Also, our protocol employs a hierarchical flooding mechanism that scopes the query's secondary dissemination, thereby reducing the overhead of discovery. While not shown, our protocol marginally underperforms flooding when the comparison is restricted to successful query domain constructions.

(a) TTL=3

(b) TTL=5

(c) TTL=7

Fig. 7. Average number of messages transmitted per query vs. the number of sensor nodes in the network.
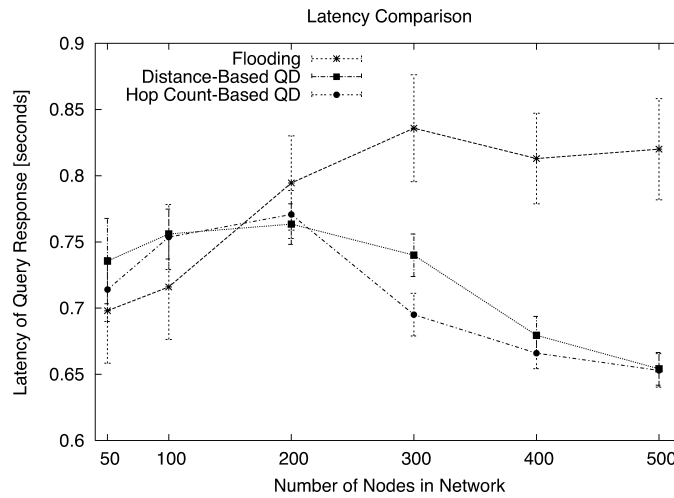
Fig. 8. Average query latency vs. number of sensor nodes in the network.

However, the flooding protocol finds far fewer query domains on its first try, and therefore this occurrence is significantly less frequent. In addition, even when the comparison is restricted only to successful query domains, the flooding based protocol outperforms our protocol only when the overhead associated with beaconing is not accounted for.

Figure 8 shows the average query response time for the protocols. Initially, our protocol's performance is similar to flooding, but the benefits increase significantly as node density increases. The added overhead of a complete flooding approach causes contention in the network, resulting in delayed responses. In addition, the query domain construction allows nodes to cooperate, enabling faster, more local results. Consequently, the average path lengths of our protocols are smaller than those of flooding-based protocol leading to faster response times. The average path lengths for our protocols vary from 2.8 to 4.9 hops, while the average path length for flooding varies from 3.6 to 6.4 hops. The results presented in this graph are based the 802.11 MAC employed by the OMNeT++ simulator. The latency in TOSSIM is slightly higher because the 802.11 MAC layer is always ready to transmit unlike some sensor network specific MAC implementations. However, the relationship between the different lines in the graph is similar and the insights gleaned are the same.

Figure 9 shows the percentage of queries successfully resolved as the queries become increasingly complex. Every point in the graph represents 500 runs in the simulation. This metric determines whether it is effective to rely on collaborative behavior as opposed to simply searching for powerful nodes. To increase the complexity, we varied the number of sensor types required by the query from two to five. Figure 9 shows that as the number of sensing units required for a single query increases, the probability of finding a satisfactory sensor or set of sensors drops substantially. In the graph, we do not distinguish between the underlying causes that lead to the failure to locate a valid query domain. Lack of valid query domains in the system to begin with, collision
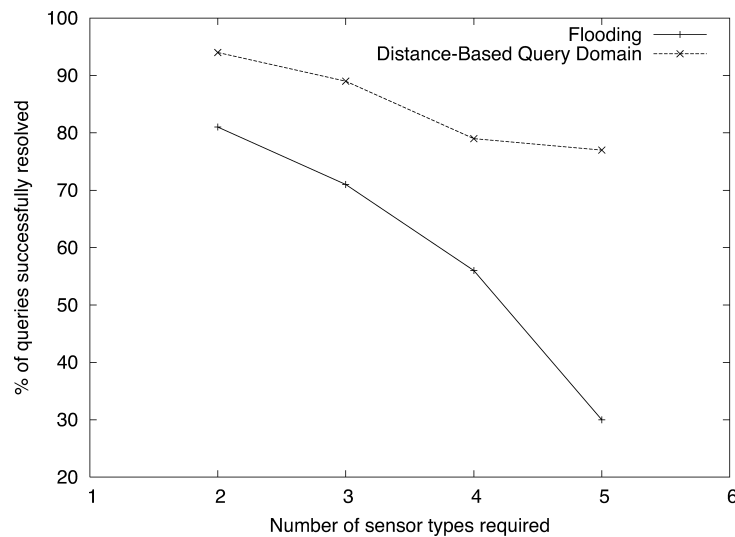
Fig. 9.   Percentage of successfully resolved queries.

related packet losses, and the conservativeness of the protocol can all result in failure to detect a valid domain. In practice, it is difficult to determine the exact cause of failure. However, by using simulator logs we were able to eliminate those queries that did not have a valid domain in the network to begin with. When networks with only valid query domains are used, the success percentage of our protocol is significantly higher. For example, its success percentage for a query with five sensors (the most complicated query in our tests), rises to 97.6% when it is guaranteed that the underlying network had valid query domains. This reaffirms that the query domain facilitates the collaboration of several less powerful nodes for complex tasks.

6.4 Persistent Protocol Results

A common mode of querying sensor networks is to continuously request data. For example, one might like to know the weather conditions periodically over an extended time interval. To accomplish this efficiently, our protocol allows *persistent* queries. The first part of the persistent protocol is identical to the protocol above and returns a path connecting the query issuer to the query domain. After the query domain is established, each node in the return path keeps track of its parent and its children. The query issuer has no parent, and the last node in the path has no children. When the query issuer needs to use the query domain multiple times, it simply sends the message only to its children, who in turn forward it to their children. To maintain group consistency, nodes periodically send beacons to their parents and children. The beacon frequency may vary depending on node mobility and application requirements; a high beacon frequency allows the nodes to discover changes in the query domain configuration faster. If a node does not hear from its parent or one of its children for a specified time interval (three beacon intervals), it disqualifies itself from
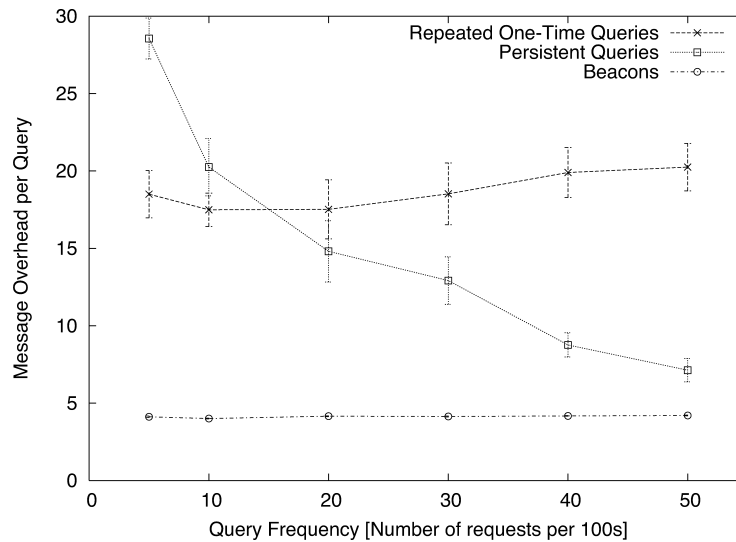
Fig. 10.   Message overhead per query as a function of query frequency.

the query domain, which eventually leads to the destruction of the entire query domain. The query issuer must then reissue the original query, constructing a new query domain.

Since we address pervasive computing environments where users typically move through sensor networks, we tested our persistent query protocol by simulating a moving query issuer. The query issuer moves at a rate of 10 feet every 25 seconds in a random fashion over a 100-second window. This corresponds to a relaxed walk. The beacon frequency was set to a relatively low value of one every five seconds. We issued persistent queries with different frequencies in a static network, changing the rate at which a query issuer makes a request from 10 to 50 times in a 100-second window. We compared this to issuing the same number of repeated one-time queries. Once again, we use a query where the number of required sensors is set to four and the reference function scopes the query to five hops.

Figure 10 shows the behavior of the persistent query in comparison to repeated uses of one-time queries as the query issuer moves. We compared the number of responses that the querier gets for both the one-time and the persistent query and ensured that they are similar. In both cases, at least 80% of the queries had responses regardless of mobility. Figure 10 shows the total overhead for all nodes in the network for persistent queries and the fraction that beaconing accounts for. At low query frequencies (the left side of the figure), the query domain is set up and maintained, even though the query issuer is likely to move before the query domain is used a second time. Therefore, the expense of maintenance is not recouped over time. Using a series of repeated one-time queries, instead, leads to lower overhead. However, as the frequency with which queries are issued over a persistent query domain increases, the overhead per query decreases. This is because more queries can be successfully resolved before

the query issuer moves and the query domain breaks. Consequently, the cost of maintaining the query domain is amortized over these queries. The use of persistent queries shows similar benefits for static networks. In fact, for static networks, the beaconing frequency can be set to a lower value because query domains are unlikely to break. Thus the benefit of using persistent queries is even greater for less dynamic networks.

These results show that the query domain abstraction scales well as the number of nodes in the network increases both when the network is static and when there is some mobility. The communication overhead and latency are favorable compared to alternatives. When there is a need for persistent queries, our persistent protocol scales well.

6.5 Query Domain Conservativeness Results

In Section 5, we detailed ways in which our protocol was conservative with respect to the formalization stated earlier. Here, we evaluate how conservative it is in practice. We determine how conservative our approach is by evaluating how well our protocol performs with respect to what is ideally possible. There are two metrics of relevance: the percentage of valid query domains that our protocol fails to detect and the number of times it fails to identify a single query domain when there are in fact valid query domains in the network. The former gives a measure of whether our protocol can find the majority of the query domains. If the user is interested in choosing a particular query domain from a selection, the number of query domains available at his disposal may be important. The latter is a measure of how often a user-defined relationship exists in the sensor network but goes undetected by our protocol. This is important in scenarios like crop failure detection, where not detecting a query domain may result in high monetary loss.

One aspect that impacts the success of our protocol is the proximity function used during domain construction. Since the proximity function is an important component of a query that the user is likely to vary, we varied its threshold and studied its impact on the percentage of query domains identified by our protocol. The distance proximity threshold was varied from 10 to 50 feet in a network where the number of nodes was set to 100. A threshold of 10 feet represents a strict proximity function, while a higher value like 50 feet indicates a far more relaxed constraint. While the simulations below are specific to the nature of the proximity function used, we tested them with other proximity functions and the insights gained from these simulations hold true in general. The query issued in the experiments in this section requires that the query domain contains three different sensor types. We did not employ a reference function for this experiment.

Figure 11 shows how our protocol compares against an ideal protocol. We see that, regardless of the threshold, at least 55% of all possible query domains are detected by our protocol. When the distance threshold is very small (left side of the figure), reflecting a very strict proximity function, the only possible domains are those that can be satisfied by the same node or nodes very close to one another. For the most part, these would in fact be connected in a
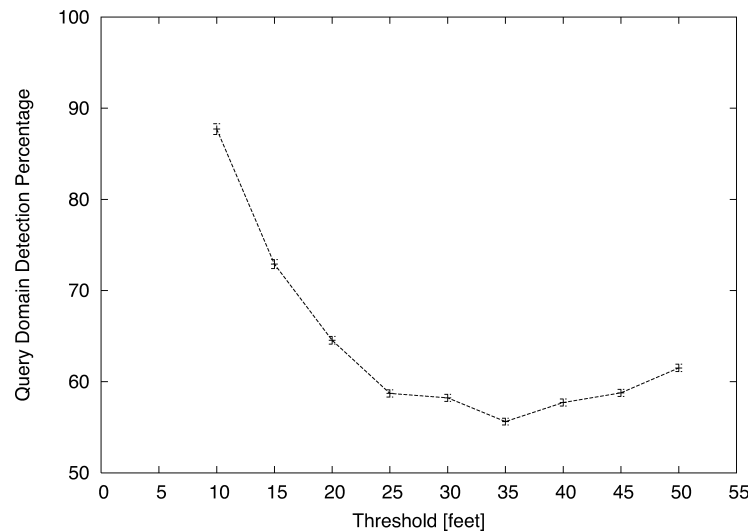
Fig. 11.   Success percentage of query domain detection as a function of the distance threshold.

single path which our protocol is very likely to detect. However, as the proximity function's threshold is increased, the success percentage decreases. This is because the nodes that are relatively far from one another can now be a part of a valid domain, and the likelihood that these nodes are connected through topologies other than single network paths increases. As explained in Section 5, some of these query domains cannot be identified by our protocol. As the figure demonstrates, no matter how relaxed the threshold of the proximity function is, the query domain protocol detects a large subset of the query domains in the network.

Another factor that can significantly impact the success of the query domain protocol is the node density. We varied the number of nodes in the network from 50 to 300. The percentage of query domains detected shows a trend very similar to the one obtained by varying the threshold in Figure 11. As the network density increases, the percentage of query domains detected decreases but the number of query domains detected increases significantly. For example, when the number of nodes in the network is set to 300, our protocol detects 49% of all possible query domains in the network and the actual number of query domains detected is a large value of 410. Most applications will not require more than the correct detection of a few query domains. Since a very large number of valid query domains are detected, one can confidently locate a particular query domain of interest by coupling this proximity function with an appropriate reference function as demonstrated in the previous experiments. The number of times our protocol failed to detect even a single valid query domain is negligible. We counted the number of times in 1000 runs when there were valid query domains in the network but no query domain was discovered by our protocol. When the network is very sparse, one or two of the runs out of a thousand failed to generate a path that was a query domain. For example, for a 50 node network,
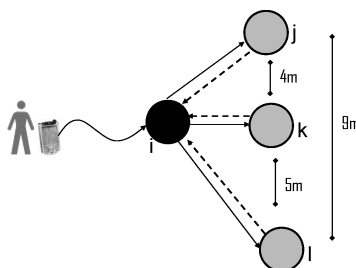
Fig. 12.   High-level coordinator protocol.

the number of such failures was just two. In connected networks (more than 100 nodes), the number of such failures was always zero.

While other protocol designs that are less conservative are possible, we determined that the simplicity and cost-effectiveness of our protocol outweighs the limitations of these alternatives. Alternate protocols may detect more domains at the expense of added complexity and cost. One alternative protocol that we considered uses a central coordinator to alleviate the conservativeness of the query domain protocol. Figure 12 shows a pictorial representation of this protocol. Consider a situation where nodes i, j, k, and l have one of the four desired sensors each, and the proximity function constrains them to be within 10m of one another. In the coordinator-based protocol, an initial flood is disseminated through the network until a node containing at least one of the given sensors is located. This node, henceforth deemed the coordinator, then proceeds to disseminate scoped floods to all its neighbors. Node i is the coordinator in Figure 12. Nodes j, k, and l receive the query from node i. Since they partially satisfy the query, they each send the sensors they satisfy and their location to the coordinator. We call such information *partial aggregates*. Upon receiving these messages, the coordinator checks to see if, by using these nearby nodes, it can satisfy the proximity function. Nodes i, j and k are clearly within 10m of the coordinator i. Otherwise, they would not have returned their partial aggregates. However, on receiving the locations of nodes j, k and l, the coordinator checks if the distance between $\{j, k\}$, $\{k, l\}$ and $\{j, l\}$ are all less than 10m. This is, in fact, the case in the figure, and the coordinator concludes that the proximity function is satisfied. A query response is returned to the querier by the coordinator. In this protocol, it is possible to receive messages from several nodes that satisfy a query partially. More than one combination of these nodes may completely satisfy the query. It is infeasible to perform a combinatorial proximity satisfaction check for every combination of nodes. Thus, we restrict the coordinator to return a query result the very first time the query is completely satisfied.

We implemented the coordinator protocol and compared it to the query domain protocol provided earlier. We varied the proximity function's threshold from 15 feet to 40 feet in a 100-foot by 100-foot simulation. The goal was to see if there was any significant benefit in using the coordinator-based protocol in terms of the number of query responses received. There was no noticeable difference. The protocols produced an almost identical number of query resolutions.
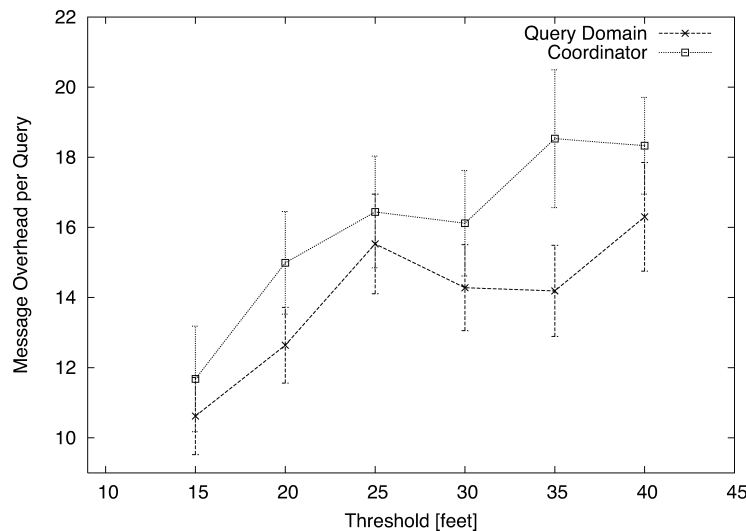
Fig. 13.   Message overhead per query as a function of the distance threshold for the coordinator protocol and query domain protocol.

This further reaffirms that the query domain protocol almost always returns a valid query domain.

Figure 13 shows the number of messages transmitted by the sensors to resolve a query for the two protocols. The message overhead associated with the query domain protocol is slightly less than the coordinator protocol. This is because the query is disseminated only along a few paths in the network, while it gets disseminated along multiple paths using the coordinator-based protocol. While the margin may vary depending on the threshold and the node density, we found that the coordinator-based protocol is typically more expensive. There was very little difference between the two protocols when latency is used as a comparison metric. These results are omitted for brevity.

These simulations show that the query domain protocol detects a large percentage of all possible query domains. In most practical sensor network applications, the user is interested in retrieving a small number of query domains. Since our protocol detects a large number of query domains, it lends itself easily to selecting an appropriate one through the proper use of reference functions. The query domain protocol is completely distributed, and variants like the persistent query domain protocol can be easily created. For the few applications that may require finding all possible query domains, the coordinator-based protocol may be used.

7. RELATED WORK

Our work is related to several different classes of systems that exist in the literature. First and foremost, the query domain is essentially a grouping mechanism. Grouping mechanisms have been well studied in the sensor network community, and can be broadly classified as neighborhood abstractions and

grouping abstractions. Second, since the purpose of our query domains is to query for data, we discuss how it relates to existing query processing systems. Third, as we use data content to form domains, our work overlaps to some of the work done on datacentric routing. Finally, our work is somewhat related to new work on macroprogramming where the emphasis is to program the network as a whole instead of individual sensors.

7.1 Neighborhood Abstraction

Neighborhood abstractions help define a group consisting of nodes and its physical neighbors. Hood [Whitehouse et al. 2004] provides a neighborhood abstraction with the goal of easily encapsulating membership, data sharing, and messaging between neighboring nodes. Subsets of nodes within physical communication range form a neighborhood. Nodes in the neighborhood exchange attributes with one another, and each node applies a filter to determine if a particular attribute is to be cached. In Abstract Regions [Welsh and Mainland 2004], the sensor network is scoped into regions defined in terms of radio connectivity or geographic location, and aggregation operations can be performed on these regions. In contrast, *Logical neighborhoods* [Mottola and Picco 2006] focuses on overlay networks defined by logical properties. Individual nodes export tagged information (such as sensor type, sensor reading, etc.) which are then compared against a neighborhood template that serves as a condition for group membership. Similarly, EnviroTrack [Abdelzaher et al. 2004] groups nodes based on a detected event. All nodes that detect the same event (e.g., identifying a car) form a group. Finally, a neighborhood abstraction system similar to ours is *Scenes* [Kabadayı and Julien 2007]. Here, a mobile device forms a *scene* around itself which allows applications to specify the required types and the conditions which limit the sensors that can participate in the scene.

A common feature of all these systems is that they provide a mechanism to form a group around the node issuing the query. In contrast, our system allows a grouping structure to be imposed in any part of the network. Our system can easily accomplish the functionality of neighborhood abstraction by simply setting the reference node to the one issuing the query. However, it can also perform grouping based on arbitrary application imposed constraints in any part of the network. Also, these approaches typically require proactive exchanges of attributes to form neighborhoods. Our approach is less restrictive because we form the query domain reactively at query time. This reduces the amount of information that needs to be programmed in the network at deployment time, facilitating general-purpose sensor network deployments.

7.2 Grouping Abstractions

Grouping mechanisms tend to focus on forming coalitions that are more general and typically contain membership criteria beyond just neighborhood constraints. A popular technique in this category is clustering. Our work is most similar to clustering algorithms that satisfy application level requirements. LEACH [Heinzelman et al. 2000], was one of the earliest cluster-based systems designed for mobile adhoc and sensor networks that used application level

communication requirements as a grouping criterion. LEACH forms groups to conserve energy and assumes that all nodes are with in communication range of the base station. By comparing the signal strengths of different nodes, the node with the best signal strength is deemed the cluster head. To prevent the cluster head from draining all its resources, the role is rotated periodically among the nodes. The Cluster Aggregation Technique (CAG) [Yoon and Shahabi 2007] also employs a clustering mechanism. CAG makes use of the fact that most sensor data exhibits high spatial and temporal correlation. The clusters are created based on the similarity of the data being transmitted. The cluster head alone transmits aggregated data, thus reducing energy consumption significantly. The primary goal of these two algorithms is to typically create a few random clusters and then rotate the leader role to keep the clusters operational for as long as possible. In contrast, we focus more on the cluster creation process. To our knowledge, no other work forms clusters based on the capabilities of heterogeneous sensors and the data values extracted from those sensors simultaneously. While our query domain can be considered a cluster of related nodes containing various capabilities, its implementation leads to a flat hierarchy typically not found in these cluster-based systems.

SpatialViews [Ni et al. 2005] is a system that allows grouping devices based on the services they offer and their location information. It is a high level programming language that allows operations to be performed on the chosen subset through code migration. This system was designed for grouping powerful devices like cameras and laptops. Its focus is on higher level constructs like language design and resource tradeoff. We focus more on group formation and do not explicitly require location information of the different devices in our group.

7.3 Query Processing Systems

TinyDB [Madden et al. 2005] and Cougar [Yao and Gehrke 2002] are two examples of systems that attempt to make the user completely oblivious to the underlying network. They view the entire sensor network as a relational database which can be queried using an SQL-style syntax. A user's query is flooded to every node in the network, and the response is sent back to the user by forming a spanning tree rooted at the sink. While the "group-by" clause provides a way to perform groupings based on symbolic information the sensors have been tagged with (in a manner similar to logical neighborhoods, above), it is not possible to specify arbitrary functions based on physical properties.

Another system that addresses a method to resolve complex queries is ACQUIRE [Sadagopan et al. 2005]. ACQUIRE provides a mechanism to resolve complex queries into a sequence of simple queries. When a subquery can be evaluated in the network, ACQUIRE invokes a d-hop look ahead to locate nodes that can resolve the subsequent subqueries. It is close to our work in the sense that multiple sensors are typically used to address each of the subqueries. However, unlike our work, ACQUIRE does not form a group while processing complex queries. For persistent queries, our abstraction can be leveraged repeatedly, reducing energy usage in resource discovery, while ACQUIRE must reestablish which sensors to use every time.

7.4 Datacentric Routing

Work in this area can be considered a subset of that done in query processing. The key insight motivating such designs is that routing should take into account the semantic information of the application data instead of relying on node or router identification information. In Directed Diffusion [Intanagonwiwat et al. 2000], one of the earliest systems in this area, the data generated at each sensor node is stored in attribute-value pairs. The query issuer injects interests into the network. Nodes that contain data satisfying the interest send their data back to querier. Directed diffusion assumes that a single powerful node can resolve all the interests that a user is interested in. As we have shown in this article, it is impractical to assume that a single node will be able to answer all the query's interests. We extend the power of data centric routing with the ability to form complex groups at runtime making use of the inherent heterogeneity available in the system.

7.5 Macroprogramming Abstractions

Finally, systems that simplify programming of sensor networks are related to our work to some degree. Kairos [Gummadi et al. 2005] and Regiment [Newton and Welsh 2004] employ a macroprogramming approach in which a user writes a single program, which is then distributed across the sensor network. Kairos provides abstractions to facilitate this task, while Regiment focuses on investigating the suitability of functional programming to the sensor network domain. Kairos imposes a graph structure on nodes that should be involved in the program (which is similar to the path list generated for our query domain). These techniques focus on the feasibility of automating mundane tasks like communication using routing trees, to reduce the time and effort required to deploy sensor network applications. They do not address the issues of querying and grouping mechanisms that are the main focus of this work.

8. CONCLUSIONS AND FUTURE WORK

We have described the query domain abstraction that enables the best-suited coalition of sensor nodes to answer a particular query. Since the specification of the proximity function to form the query domain does not require any broadcasts a priori, it reduces the communication overhead and supports multipurpose sensor networks. We have related the abstraction, its protocol implementation, and performance evaluation demonstrating the scalability of the protocol with increasing number of nodes and in the face of mobility. We have demonstrated the flexibility of the abstraction by extending it to incorporate persistent queries. We also implemented different protocols to account for different degrees of correctness that the application may require. In general, the protocol presented in this article should help in the deployment of a large class of sensor network applications.

Future work will investigate the feasibility of injecting arbitrary code into a network to enhance the expressiveness of the proximity function. Currently, we have a series of library functions available to the protocol to calculate distance, latency, check for tagged information, etc. While this covers a large subset of

proximity functions, we will investigate means to create arbitrary relationships more easily. In addition, the proximity function as described here allows the application to incorporate the properties of its choosing but the protocol itself does not make any assumptions about the underlying characteristics of the network that may ultimately affect the performance of the application. Future work will look at building some traffic engineering techniques critical to the efficiency of sensor networks (such as overall battery power conservation, load balancing, etc.) into query domain construction. We will also investigate how many query domains should be realistically kept in the querier's memory and how to deal with issues like group overlap if the query issuer chooses to deal with multiple query domains simultaneously.

REFERENCES

ABDELZAHER, T., BLUM, B., CAO, Q., CHEN, Y., EVANS, D., GEORGE, J., GEORGE, S., GU, L., HE, T., KRISHNAMURTHY, S., LUO, L., SON, S., STANKOVIC, J., STOLERU, R., AND WOOD, A. 2004. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems*. 582–589.

GUMMADI, R., GNAWALI, O., AND GOVINDAN, R. 2005. Macro-programming wireless sensor networks using Kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems*. 126–140.

HAMMER, J., HASSAN, I., JULIEN, C., KABADAYI, S., O'BRIEN, W., AND TRUJILLO, J. 2006. Dynamic decision support in direct-access sensor networks: A demonstration. In *Proceedings of the International Conference on Mobile Ad-hoc and Sensor Systems*. 578–581.

HEINZELMAN, W., CHANDRAKASAN, A., AND BALAKRISHNAN, H. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the International Conference on System Sciences*. 8020–8029.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. *SIGPLAN Notices 35*, 11, 93–104.

INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking*. 56–67.

KABADAYI, S. AND JULIEN, C. 2007. A local data abstraction and communication paradigm for pervasive computing. In *Proceedings of the International Conference on Pervasive Computing and Communications*. 57–68.

KRISHNAMURTHY, L., ADLER, R., BUONADONNA, P., CHHABRA, J., FLANIGAN, M., KUSHALNAGAR, N., NACHMAN, L., AND YARVIS, M. 2005. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *Proceedings of the Conference on Embedded Networked Sensor Systems*. 64–75.

LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the Conference on Embedded Networked Sensor Systems*. 126–137.

LIANG, O., SEKERCIOGLU, Y., AND MANI, N. 2007. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the Wireless Communications and Networking Conference*. 3495–3500.

MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Datab. Syst. 30*, 1, 122–173.

MAINWARING, A., POLASTRE, J., SZEWCZYK, R., CULLER, D., AND ANDERSON, J. 2002. Wireless sensor networks for habitat monitoring. In *Proceedings of the International Workshop on Wireless Sensor Networks and Applications*. 88–97.

MENDELSOHN, R. 2007. What causes crop failure? *Climatic Change 81*, 1, 61–70.

MILLMAN, G. 2004. Virtual vineyard. *Accenture Outlook Journal*.

MOTTOLA, L. AND PICCO, G. 2006. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the International Conference on Integrated Internet Ad-Hoc and Sensor Networks*.

NEITZEL, R., SEIXAS, N., AND REN, K. 2001. A review of crane safety in the construction industry, applied occupational and environmental hygiene. *Appl. Occupat. Environ. Hyg. 16*, 12, 1106–1117.

NEWTON, R. AND WELSH, M. 2004. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the Workshop on Data Management for Sensor Networks*. 78–87.

NI, Y., KREMER, U., STERE, A., AND IFTODE, L. 2005. Programming for ad-hoc networks of mobile and resource constrained devices. In *Proceedings of the Conference on Programming Language Design and Implementation*. 249–260.

SADAGOPAN, N., KRISHNAMACHARI, B., AND HELMY, A. 2005. Active query forwarding in sensor networks (ACQUIRE). *Elsevier Ad Hoc Netw. 3*, 1, 91–113.

WELSH, M. AND MAINLAND, G. 2004. Programming sensor networks using abstract regions. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

WHITEHOUSE, K., SHARP, C., BREWER, E., AND CULLER, D. 2004. Hood: A neighborhood abstraction for sensor networks. In *Proceedings of the Conference on Mobile Systems*. 99–110.

YAO, Y. AND GEHRKE, J. 2002. The Cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record 31*, 3, 9–18.

YOON, S. AND SHAHABI, C. 2007. The Clustered aggregation (CAG) technique leveraging spatial and temporal correlations in wireless sensor networks. *ACM Trans. Sens. Netw. 3*, 1, 3.