Abstract

This article introduces a sensor information networking architecture, called SINA, that facilitates querying, monitoring, and tasking of sensor networks. SINA serves the role of middleware that abstracts a network of sensor nodes as a collection of massively distributed objects. SINA's execution environment provides a set of configuration and communication primitives that enable scalable and energy-efficient organization of and interactions among sensor objects. On top the execution environment is a programmable substrate that provides mechanisms to create associations and coordinate activities among sensor nodes. Users then access information within a sensor network using declarative queries, or perform tasks using programming scripts.

Sensor Information Networking Architecture and Applications

Chien-Chung Shen, Chavalit Srisathapornphat, and Chaiporn Jaikaeo, University of Delaware

The advent of technology has facilitated the development of small, low-power devices that combine programmable general-purpose computing with multiple sensing and wireless communication capabilities. Composing these sensor nodes into sophisticated ad hoc computational and communication infrastructures to form sensor networks will have significant impact on applications ranging from military situation awareness to factory process control and automation [1].

The sheer number of sensor nodes and the dynamics of their operating environments (e.g., limited battery power and hostile physical environment) pose unique challenges in the design of sensor networks and their applications. Issues concerning how information collected by and stored within a sensor network could be queried and accessed and how concurrent sensing tasks could be executed internally and programmed by external users are of particular importance. In this article we describe a sensor information networking architecture, called SINA, that facilitates querying, monitoring, and tasking of sensor networks. The following section describes the components and information abstraction of the architecture. An implementation of the architecture, including the sensor programming language called Sensor Query and Tasking Language (SQTL) and its execution environment, is described as well. We then introduce data gathering operations for sensor information, and describe issues related to interworking between mobile users and stationary sensor nodes. Sample applications to illustrate the capability of the information gathering operations and SQTL are also presented along with their simulation studies.

SINA — A Middleware Architecture

Conceptually, a sensor network is modeled as a collection of massively distributed objects. SINA plays the role of middleware, allowing sensor applications to issue queries and command tasks into, collect replies and results from, and monitor changes within the networks (Fig. 1a). SINA modules, running on each sensor node, provide adaptive organization of sensor information, and facilitate query, event monitoring, and tasking capability (Fig. 1b).

In contrast to conventional distributed databases in which information is distributed across several sites, the number of sites in a sensor network equals the number of sensors, and the information collected by each sensor becomes an inherent part (or attributes) of that node [2]. To support energy-efficient and scalable operations, sensor nodes are autonomously clustered. Furthermore, the data-centric nature of sensor information makes it more effectively accessible via an attribute-based naming approach instead of explicit addresses [1]. SINA architecture consists of the following functional components.

Hierarchical Clustering — To facilitate scalable operations within sensor networks, sensor nodes should be aggregated to form clusters based on their power levels and proximity (Fig. 2a). The aggregation process could also be recursively applied to form a hierarchy of clusters (Fig. 2b). Within a cluster, a cluster head will be elected to perform information filtering, fusion, and aggregation, such as periodic calculation of the average temperature of the cluster's coverage area. In addition, the clustering process should be reinitiated should the cluster head fail or run low on battery power. In situations where a hierarchy of clusters is not applicable, the system of sensor nodes is perceived by applications as a one-level clustering structure, where each node is a cluster head by itself. The clustering algorithm introduced in [1] allows sensor nodes to automatically form clusters, elect and re-elect cluster heads, and reorganize the clustering structure if necessary.

Attribute-Based Naming — With the large population of sensor nodes, it may be impractical to pay attention to each individual node. Users would be more interested in querying which area(s) has temperature higher than 100˚F, or what is the average temperature in the southeast quadrant, rather than the temperature at sensor ID#101. To facilitate the data-centric characteristics of sensor queries, attribute-based naming is the preferred scheme [1]. For instance, the name

This research was presented in part at the International Workshop on Pervasive Computing, Toronto, Canada, August 2000.

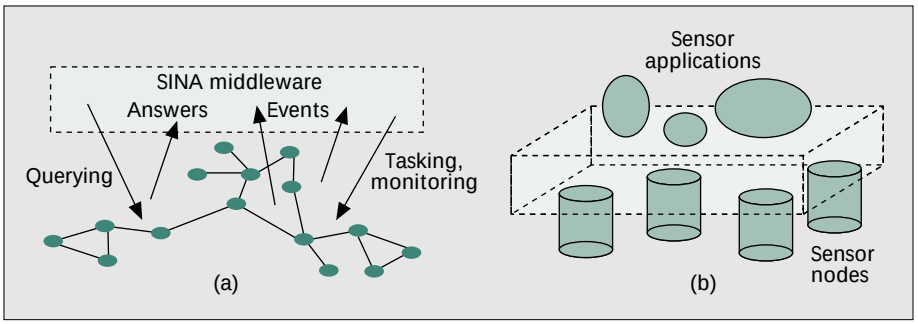

Figure 1. *A model of sensor networks and SINA middleware.*

`[type = temperature, location = N-E, temperature = 103]` describes all the temperature sensors located at the northeast quadrant with a temperature reading of 103˚F. These sensors will reply to the query "which area(s) has temperature higher than 100˚F."

Location Awareness — Due to the fact that sensor nodes operate in physical environments, knowledge about their own physical locations is crucial. Location information can be obtained via several methods. Global Positioning System (GPS) is one of the mechanisms that provide absolute location information. For economical reasons, however, only a subset of sensor nodes may be equipped with GPS receivers and function as location references by periodically transmitting a beacon signal telling their own location information so that other sensor nodes without GPS receivers can determine their approximate positions in the terrain. Other techniques for obtaining location information are also available. For example, optical trackers [3] give high precision and resolution location information but are only effective in a small region.

With integration of these three components, the following two sample queries may be carried out effectively:

- *Which area(s) has temperature higher than 100˚F?* In theory, the query is broadcast to and evaluated by every node in the network. Despite probably the best returned result, the query would suffer from long response time. In practice, each cluster head may periodically update the temperature readings of its members, and the query can now be multicast to and evaluated by cluster heads only. This results in better response time at the expense of less accurate answers. Queries under stringent timing constraints can be evaluated by cluster heads of a higher tier.
- *What is the average temperature in the southeast quadrant?* Similarly, the average temperature of each cluster can be periodically updated and cached by cluster heads. Furthermore, the query should be delivered to nodes located (named) in the southeast quadrant only.

Information Abstraction

In SINA, a sensor network is conceptually viewed as a collection of datasheets; each datasheet contains a collection of attributes of each sensor node. Each attribute is referred to as a *cell*, and the collection of datasheets of the network present the abstraction of an *associative spreadsheet*, where cells are referred to via attribute-based names. Initially, the datasheet of each sensor node contains a small number of predefined attributes. Once these sensor nodes are deployed and form a sensor network, they can be requested by other nodes (e.g., from their cluster heads) to create new cells by evaluating valid cell construction expressions that may obtain information from other cells, invoke a system-defined function, or aggregate information from other datasheets.

Each newly created cell must be uniquely named and becomes a node's attribute, which can be either a single value (e.g., remaining battery power) or multiple values (e.g., history of temperature changes in the past 30 minutes). By incorporating a hierarchical clustering mechanism and an attribute-based naming scheme, the architecture provides a powerful set of operations to deal with data access and aggregation among sensor nodes. The mechanism of *associative broadcast* [4] has been employed to facilitate node interaction via attribute-based naming.

Sensor Query and Tasking Language

As part of the architecture, SQTL [5] plays the role of a programming interface between sensor applications and the SINA middleware. It is a procedural scripting language, designed to be flexible and compact, with a capability to interpret simple declarative query statements. In addition to sensor hardware access (e.g., `getTemperature`, `turnOn`), location-aware (e.g., `isNeighbor`, `getPosition`), and communication (e.g., `tell`, `execute`) primitives, it also provides an event handling construct, which is suitable for sensor network applications where sensor nodes are programmed to process asynchronous events such as receiving a message or an event triggered by a timer. By using the **upon** construct, a programmer can create an event handling block accordingly. Currently, three types of events are supported by SQTL:

- Events generated when a message is received by a sensor node
- Events triggered periodically by a timer
- Events caused by the expiration of a timer

These types of events are defined by the SQTL keywords **receive**, **every**, and **expire**, respectively.

An SQTL message, containing a script, is meant to be interpreted and executed by any node in the network. In order to target a script to a specific receiver or group of receivers, the message is to be encapsulated in an *SQTL wrapper* which acts as a message header to indicate the sender, the receivers, the particular application running on the receivers, as well as parameters for the application.

We adopt the syntax of the Extensible Markup Language (XML) for the SQTL wrapper which defines an application layer header that is capable of specifying a complicated addressing scheme for attribute-based names. Table 1 summarizes common SQTL wrapper fields.

The Sensor Execution Environment

A sensor execution environment (SEE), running on each sensor node, is responsible for dispatching incoming messages, examining all arrival SQTL messages, and performing the appropriate operation for each type of action specified in the messages. SEE looks inside the `receiver` argument of a message and decides, based on its value, whether to forward the message to the next hop. Messages with ALL_NODES in their group sub-

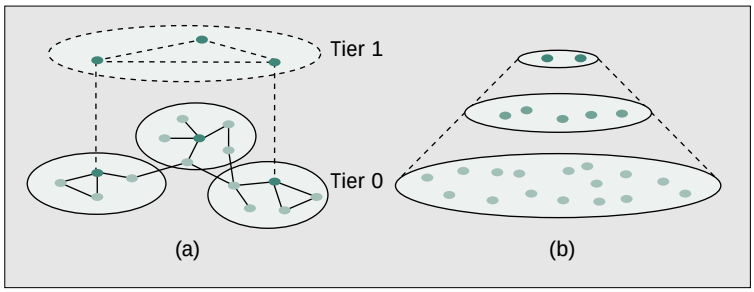

Figure 2. *a) Clustering and b) a cluster hierarchy.*

Argument	Meaning
sender	The sender of an SQTL message wrapper
receiver group criteria	Potential receivers specify by two subarguments: Subargument of receiver to specify group of receiver; its possible value can be one of ALL_NODES, or NEIGHBORS Subargument of receiver to specify selection criteria of receivers
application-id	A unique ID for each application in the same sensor network
num-hop	Number of hops away from a gateway node
language	Specify a language used in content
content	A payload containing a program, a message, or return values
with (optional) parameter type name value	Tuples of parameters used in the program passed from sender to receiver Repeatable subargument of with Data type of the parameter Name of the parameter Value of the parameter

Table 1. *Arguments used by actions in an SQTL wrapper.*

arguments will be rebroadcast to every sensor node in the network, and those with NEIGHBORS will only be forwarded to the nodes' one-hop-away neighbors. An attribute-based name in the form of a list of attribute-value pairs indicated by the `criteria` field will be compared with the receiver's attributes stored in its datasheet. SEE only accepts the message if the node's attributes satisfy the criteria. This process of matching a message with its potential receiver(s) when arriving at the receiver(s) is termed *late binding* and is described in [4].

Once an SQTL script is injected from the *front-end* node (a special node directly connected to the network) to one or more sensor nodes, the script may push itself to other nodes in order to complete the assigned task. A `tell` message is then generated after a result is produced at each individual sensor node and is delivered back to the requesting node, which is normally the upstream node from which the script came. Figure 3 depicts the dispatching of incoming messages performed by SEE.

In addition to demultiplexing incoming SQTL messages, SEE also takes care of outgoing SQTL messages from all running applications. Outgoing messages will be distributed to target node(s) specified in the receiver argument through the underlying communication mechanism. SEE may perform a translation of an attribute-based name into a unique, numeric link-layer address where applicable. Otherwise, broadcast will be used at the link layer.

Built-In Declarative Query Language

For applications that collect sensor information, a user may choose to invoke the built-in query interpreter instead of explicitly writing a procedural SQTL script. The query language has been adapted from the Structured Query Language (SQL) to serve as the primary mechanism for querying sensor networks. The following sample query statement, as delivered to all cluster heads in the network (encapsulated in the SQTL wrapper), would ask every cluster head to create a new cell called *avgTemperature* which maintains the average temperature among all of its cluster members:

SELECT avg(*getTemperature()*)
AS *avgTemperature*
FROM CLUSTER-MEMBERS

As soon as an SQTL message containing such a query statement is received by target nodes, the corresponding SEE will pick the most appropriate data dissemination method available to evaluate the query.

Database techniques, such as view composition, materialization, and maintenance, are being investigated and adapted to maintain consistency among associated cells. Related research on querying a sensor network modeled as a device database may be found in [6].

Information Gathering Methods

For applications to take full advantage of the SINA architecture, an underlying communication mechanism among sensor nodes plays an important role. By providing efficient data dissemination and information gathering supports suitable for specific application requirements, SINA abstracts the low-level communications away from high-level sensor applications. When users submit queries, it is not required to explicitly define how the information will be collected inside the network. The SINA architecture selects the most appropriate data distribution and collection method based on the nature of queries and current network status. Upon receiving users' queries, the front-end node has the responsibility to interpret and evaluate the query by requesting information from other nodes. With the sheer number of sensor nodes, collisions resulting from a large number of responses propagated back to the front-end node during a short period of time create the *response implosion problem* [2] depicted in Fig. 4a. The objective of the information gathering mechanisms is to maximize the quality of responses in terms of their number and responsiveness, while minimizing network resource consumption.

Three primitive methods are introduced to accomplish the information gathering task: *sampling operation*, *self-orchestrated operation*, and *diffused computation operation*.

Sampling Operation — For certain types of applications, such as finding the average temperature over the entire network area, responses from every sensor node may cause response implosion. To reduce the degree of the problem, some sensor nodes may not need to respond if their neighbors do. Nodes make autonomous decisions whether they should participate in this application based on a given response probability, as shown in Fig. 4b.

An enhancement can be made to this approach if sensor nodes are not evenly distributed over the area. To prevent having more responses from denser areas, the response probability will be computed at each cluster-head node based on

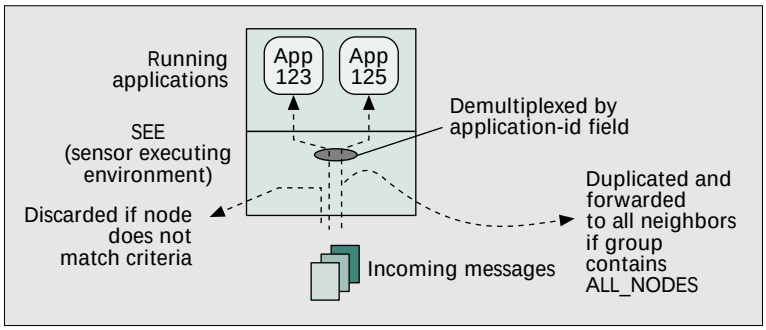

Figure 3. *Dispatching of messages received by a sensor node.*

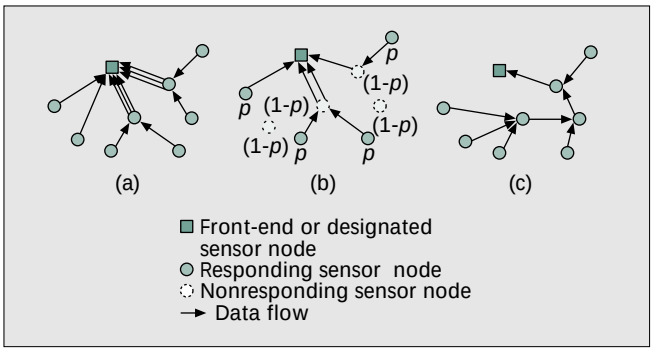

Figure 4. *a) The response implosion problem; b) number of responses reduced by assigning sensor nodes a probability* p *to answer the request; and c) diffused computation operation allowing data aggregation at intermediate nodes.*

the number of replies required from each cluster. We call this operation *adaptive probability response* (APR).

Self-Orchestrated Operation — In a network with a small number of nodes, responses from all nodes are necessary for the accuracy of the final result. Another approach to avoiding the response implosion problem is to let each node defer its sending of response(s) for a certain period of time. Despite some extra delay, this method aims to improve the overall performance by reducing the chances of collision. This operation is modified from the scheduled response approach described in [7]. Assume that nodes are distributed uniformly within the network terrain; therefore, the number of nodes within h hops away from the front-end node is proportional to h^2. The delay period at every node can be defined as

$$\text{Delay} = KH\,(h^2 - (2h - 1)r),$$

where h is the length in number of hops away from the front-end, r is a random number such that $0 < r \leq 1$, and H is a constant reflecting estimated delay per hop. To incorporate potential effects from queuing and processing delays, K is used as a compensation constant. Normally, K and H are combined and used as adjustable parameters.

Diffused Computation Operation — For this operation, each sensor node is assumed to have knowledge of its immediate communicating neighbors only. Algorithms used for gathering information are constrained by the capability that each node can only communicate to other nodes in its surrounding area. Information aggregation logic is programmed in SQTL scripts and disseminated among sensor nodes so that they know how to aggregate information en route to the front-end. The conceptual data flow is depicted in Fig. 4c. Since data got aggregated at intermediate nodes on the way back to the front-end node, the consumption of valuable network bandwidth is considerably reduced and the response implosion problem is alleviated. However, for large sensor networks, this diffusion approach might take longer to deliver results back to the front-end.

The hierarchical structure enabled by SINA allows different information gathering methods to be deployed in different levels within one application in order to optimize overall performance. The effect of the integration is described in the following sample applications. In addition, SPIN [8], a negotiation-based data dissemination mechanism, can be applied in SINA. SPIN relies on exchanging meta data before deciding to broadcast all the information. This negotiation process reduces network bandwidth usage. By integrating SPIN into SINA, the architecture will achieve a higher level of resource conservation.

Interworking Between a Mobile User and a Stationary Network

Consider the scenario where a mobile user issues a query into a stationary sensor network (at one particular sensor node called a *query resolver*) while marching forward. When the reply becomes available at the resolver, the user's point of contact with the stationary sensor network may have changed. Two mechanisms to forward the reply are proposed:

- The first mechanism requires the mobile user to constantly update his/her current point of contact with the resolver (Fig. 5a). Since all updates have to traverse all the way from the user to the resolver, this approach increases traffic load in the sensor network.
- To reduce traffic load, the mobile user may place his/her footprints along the way by sending periodic advertisements informing nearby sensor nodes. A logical link is established to the last contact point upon receiving an advertisement (Fig. 5b). With this modification, all updates are made locally related to the current position. This is known as *progressive footprint chaining.*

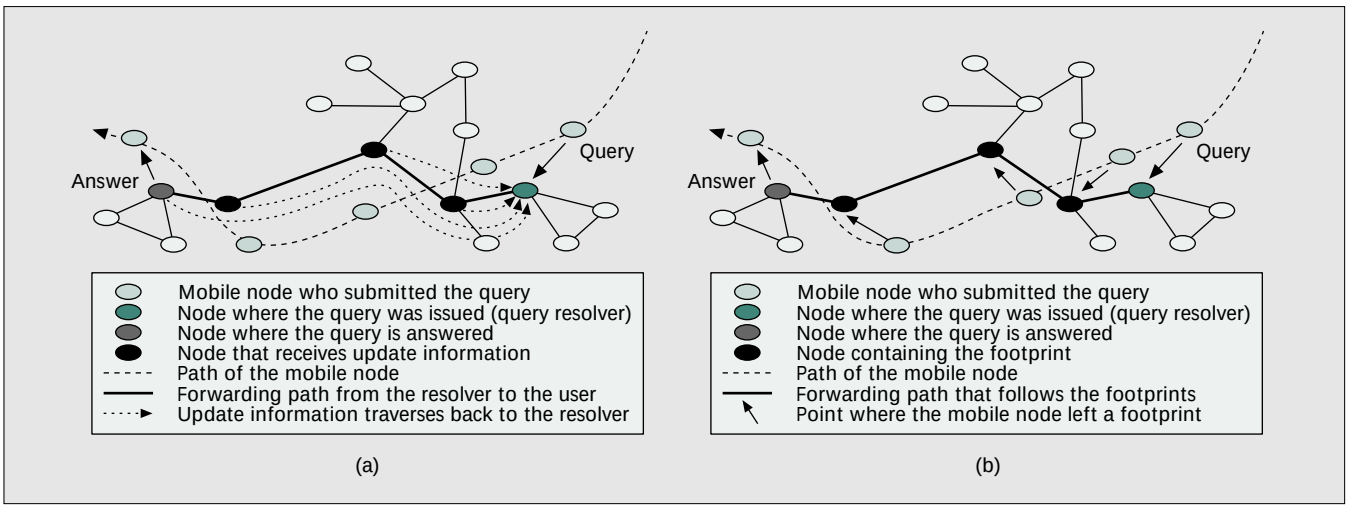

Figure 5. *Interworking between a mobile user and fixed sensors. a) The user periodically updates his/her location with the resolver; b) the user places his/her footprints while he/she is marching.*

```
CentralizedDiagnose(replyProb, kh)
  rebroadcast this message to all neighbors;
  prevNode ← message sender node;
  if uniformrandom(0, 1) < replyProb then
    numHops ← number of hops this message traversed;
    delay ← kh × [numHops² – (2 × numHops—1) × uniform random(0,1)];
    wait for delay;
    read power level and position, then send them back via prevNode;
  end if
  Upon receiving a return message, relay it back to prevNode;
```

Algorithm 1. *Centralized operation with sampling and self-orchestrated operation.*

```
AprDiagnose(ENRC)
  Rebroadcast this message to all neighbors;
  prevNode ← message sender node;
  if this is a cluster head then
    prob ← ENBC/# of children;
    construct a script requesting all cluster members to return value with
    probability prob;
  end if
  Upon receiving a return message, relay it back to prevNode;
```

Algorithm 2. *Adaptive probabilistic response operation.*

Sample Applications

To illustrate the applicability of the architecture to querying and tasking of sensor networks, we present two applications and their simulation studies using GloMoSim. The simulated sensor network environment has the following assumptions:

- All sensor nodes are stationary, and the sensor network is not partitioned.
- All sensor nodes are homogeneous in their capabilities.
- All communications are symmetric.
- No sensor nodes fail during the time the algorithms are being executed.
- The network is not expected to have routing support provided by the network layer. However, an application is able to keep track of the sender's address and to specify it as a receiver to forward results back to the sender.

Diagnosis of Sensor Networks — We define *sensor network diagnosis* to be the process of querying the status of a sensor network and figuring out the problematic (group of) sensor [9]. In order to monitor the status of a sensor network, one approach is to query as much information from as many sensor nodes as possible, and deliver the raw information to the manager for further processing. An example of employing this technique is when a manager wants to know the remaining power level within the network. In addition, to examine the correctness of results obtained from one sensing device, one possible method is to use the average of results obtained from other neighboring sensor nodes as a standard base to compare and diagnose the devices in doubt, given that the average has its deviation within an acceptable range. An example of using this method is to figure out which sensor node contains a faulty temperature sensing device.

We evaluate three different information gathering operations for diagnosis:

- Centralized with sampling and self-orchestration
- APR
- Diffused computation

Algorithm 1 describes the pseudocode for centralized operation with sampling and self-orchestrated operations. APR without self-orchestrated operation is presented in Algorithm 2. The pseudocode for sensor network diagnosis using diffused computation operation is given in Algorithm 3.

Simulation Setup — Our simulated network consists of 1024 stationary sensor nodes distributed in a grid pattern with grid units equal to 3 m, covering an area of size 100×100 m^2. Each node is equipped with a radio transceiver which is capable of transmitting a signal up to 5 m over a 2 Mb/s wireless channel. Each transmission then covers approximately eight immediate neighbors. The data link layer uses the 802.11 protocol, while the network layer employs only the IP fragmentation feature to communication with immediate neighbors without any routing protocol. All diagnostic applications are running on top of UDP. Each node is supplied a battery with enough power to at

	Total number of responses received	Fraction of expected responses received	Average response rate responses/s)	Number of MAC packets per response
Centralized approach (Prob[1] = 0.75)	229.0	29.8%	109.00	208.87
Self-orchestrated centralized (Prob = 0.75, *KH*[2] = 4 ms)	430.0	55.9%	107.50	138.17
Adaptive probabilistic response (ENRC[3] = 4)	183.8	40.4%	108.17	164.19
Diffused computation (Timeout[4] = 70 ms)	1016.0	99.2%	5080.00	14.70
Diffused computation with sampling (Timeout = 70 ms, Prob = 0.75)	767.0	99.8%	3068.00	12.78
Diffused Computation with Self-orchestration (Timeout = 70 ms, *KH* = 4 ms)	975.0	96.5%	3956.00	14.24

[1] Response probability. [2] Estimated hop delay and compensation. [3] Expected number of responses per cluster [4] Confirmation timeout

Table 2. *Experimental results from running different diagnosis operations.*

```
Diffused Diagnose(timeout)
  confirmCount ← 0;
  prevNode ← message sender node;
  send a confirm to prevNode;
  rebroadcast this message to all neighbors;
  set timer for timeout period;
  while not timeout do
    if receive a message of type confirm then
      confirmCount ← confirmCount + 1;
    end if
  end while
  answerList ← getPowerLevel();
  while confirmCount ≠ 0 do
    if receive a message of type return then
      insert the returned value into answerList;
      confirmCount ← confirmCount – 1;
    end if
  end while
  return answerList back to prevNode;
```

Algorithm 3. *Diffused computation operation.*

least make it able to carry out a complete query operation. Furthermore, for those information gathering methods that require clustering support such as APR, we assume that a clustering algorithm has completed in advance, so each node should have clustering information about its parent and children prior to the diagnosis. In the simulation we manually configure them into clusters of nine sensor nodes with a cluster head located at the center of each cluster. Once the simulated network starts and becomes ready, one node in the network, designated the front-end, will be requested to gather sensor information. This node will propagate the request throughout the network according to different diagnosis methods used.

Results and Analysis — The experimental results shown in Table 2 present four performance metrics for each of the selected diagnosis operations listed in the leftmost column. The first metric, *total number of responses* received, gives an idea of how many nodes were effectively participating during the course of diagnosis. The reason some of these numbers are not rounded is because they were obtained from running the experiment several times and calculating the averages. The next metric is the *fraction of expected received responses*. It represents a percentage amount of responses with respect to the expected number from the operation's settings. For example, we expected to see 768 responses (75 percent of 1024) received at the frontend, but there were only 229 responses, which is 29.8 percent of 768. The metric *average response* rate indicates the responsiveness of each operation in terms of the number of responses received per second, measured from the time the first response arrives until the last response is received. The last measurement is the number of medium access control (MAC) packets transmitted per response received, which is meant to show the efficiency of each technique by giving the amount of network bandwidth utilized (the lower, the better) to obtain one response.

Due to the large amount of collisions caused by the lack of response scheduling, the centralized approach without self-orchestration performs badly by all means. The actual responses received

```
<execute>
    <sender> FRONTEND </sender>
    <receiver>      <group>NODE[0] </group>
                    <criteria>TRUE </criteria>
    </receiver>
    <application-id>118 </application-id>
    <rum-hop>0 </rum-hop>
    <language>SQTL </language>
    <with>
        <parameter type="clocktype"   name="trackingTime"     value="600" />
        <parameter type="clocktype"   name="reTrackingTime"   value="40" />
        <parameter type="clocktype"   name="trackingFrequency" value=" 8" / >
        <parameter type="object"      name="target"           value="Vehicle1">
    </with>
    <content><![CDATA[
        lastSensingResult = false;
        timerApplication = createTimer(trackingTime);   // instantiate a timer
        timerApplication.start();                       // turn it on
        timerReTracking = createTimer(reTrackingTime);
        execute (ALL_NODES, "TRUE", MESSAGE["content"]);   // re-broadcast
        if ((sensor1 = getMotionSensor()).turnOn())   {     // instantiate a sensor object
          upon {                                            //      and turn it on
            receive (msg) where msg["action"] == "tell" && msg["content"] ==
            "suppress": {
                sensor1.standby(); break;
            }
            every (trackingFrequency): {
                if (sensor1.detect(target)) {
                    tell (ALL_NODES,        "TRUE", "suppress");
                    tell (NEIGHBORS,        "TRUE", "retrack");
                    tell (MESSAGE["sender"], "TRUE", "found");
                    lastSensingResult = true;
                    timerReTracking.start();
                    break;
                }
                else lastSensingResult = false;
            }
            expire (timerApplication): sensor1.turnOff(); exit(O);
          }
          upon { // After one sensor node sees the vehicle
            receive (msg) where msg["action"] == "tell" && msg["content"] ==
            "retrack": {
                if (timerReTracking.expired()) {
                    sensor1.turnOn();
                    timerReTracking.start();
                }
            }
            receive (msg) where msg["action"] == "tell" && msg["content"] ==
            "found":
                tell (MESSAGE["sender"],    "TRUE",    "found");
            every (trackingFrequency): {
                if (sensor1.detect(target)) {
                    tell (MESSAGE["sender"], "TRUE",   "found");
                    if (!lastSensingResult)
                        tell (NEIGHBORS,     "TRUE",   "retrack");
                    lastSensingResult = true;
                    timerReTracking.start();
                }
                else {
                    if (lastSensingResult)
                        timerReTracking.restart();
                    lastSensingResult = false;
                }
            }
            expire (timerReTracking) : sensor1.standby();
            expire (timerApplication): sensor1.turnOff(); exit(0);
          }
        }
        else exit(1);
    ]]> </content>
</execute>
```

Figure 6. *Complete SQTL code for the coordinated vehicle tracking algorithm.*

Vehicle tracking method	Ratio of useful/total number of sensing	Number of packets sent	Normalized cost
Ordinary	249:76,800 (1:308)	16,868	1.000
Coordinated	249:8,828 (1:35)	22,691	0.179

Table 3. *Comparison of simulation results between ordinary and coordinated vehicle tracking.*

are far less than expected since a lot of packets were dropped due to buffer overflow and the limited number of retransmissions at the MAC layer. The number of MAC packets per response is good evidence for this hypothesis. The overall performance is significantly improved with the help of self-orchestration. As presented in the table, the number of responses is almost doubled, while the number of MAC packets involved is cut nearly in half. The next result is from the adaptive probabilistic approach, the only method that utilized a clustering mechanism. In the configuration, four responses were expected to be obtained from each cluster of nine members. This means 456 responses should have been received if no packets were lost. Of these, only 184 (40.4 percent) were actually received. The diffused computation method gives very good results. It effectively makes use of SINA's active programmability (via SQTL scripts) to distribute (diffuse) computation to every node. In the simulation, the diffused computation performs concatenation of responses at nodes along the way back to the front-end. The number of MAC packets employed is reduced considerably, resulting in less chance of collisions, nearly 100 percent responses, and a relatively high response rate.

The last two experiments attempt to integrate sampling and self-orchestrated mechanisms into the diffused computation technique. In the combined diffused computation and sampling method, nodes would receive requests as usual, but would respond with probability 0.75. The result is slightly improved by the fraction of expected responses received, which becomes even closer to 100 percent. It also reduces the chance of collisions in the channel, as indicated by the decreased number of MAC packets per response. With the integrated diffused computation and self-orchestration approach, nodes receiving a request will schedule themselves to send back confirmation messages to reduce the chance of collisions. Besides the reduced response rate caused by delayed confirmation, other results do not differ much from those of the pure diffused computation technique.

Coordinated Vehicle Tracking — The vehicle tracking application is to locate a specific vehicle or moving object and monitor its movement. To detect and identify an object, integrated results from more than one type of sensor (images from a camera, vibration from a seismic sensor, noise from an audio sensor, etc.) may be required. These results are to be processed and compared to the signature of the object of interest. However, our main interest is to program a coordination algorithm in the form of an SQTL script, which can be disseminated to and executed by sensor nodes. The script controls the sensor nodes to collaboratively detect the appearance of the interested object in an effective and efficient manner. We assumed that sensor nodes can obtain final processed results of detecting and identifying the tracked vehicle from the processing of combined sensing information.

A novice approach to tracking a moving object is to ask every sensor node to sense and detect the object's signature at the same time. We call this operation the *ordinary vehicle tracking method*. However, this approach may waste sensor nodes' processing cycles, and hence inefficiently utilize a network's limited power and shorten the overall network lifetime.

Our coordinated vehicle tracking algorithm, presented in Fig. 6, is based on a suppression and reinitiation mechanism to achieve good results of tracking, but consumes less network resources than the ordinary one. The main principle of the coordinated algorithm is to let the first sensor node detecting the vehicle suppress sensing activities of all other sensor nodes so the others may stand by, which results in energy conservation. Furthermore, the node will have to reinitiate sensing activities of its neighbors in order to keep track of the moving vehicle. As long as the vehicle does not move faster than the propagation of this reinitiation message, the network can still monitor the trail of the moving vehicle. The tracking process is depicted in Fig. 7 as well.

Simulation Setup — We performed a simulation study to compare the efficiency of the ordinary and coordinated vehicle tracking mechanisms. The simulated network environment is similar to the diagnosis simulation setup described previously. The changes are that grid unit is 200 m, transmission range is 380 m, and the coverage area is 6800×6800 m^2. We modeled each sensor node to have an ability to detect and identify a moving object within 200 m. When the simulated network starts, there is one vehicle moving straight from coordinate (5, 5) toward (6800, 6800) at a speed of 15 m/s. Both tracking applications start at 15 s and last for 10 min. The tracking frequency is 7.5 times/min, or a sensor node probes the moving object every 8 s. For the coordinated algorithm, after a sensor node is suppressed and later received a reinitiation via a

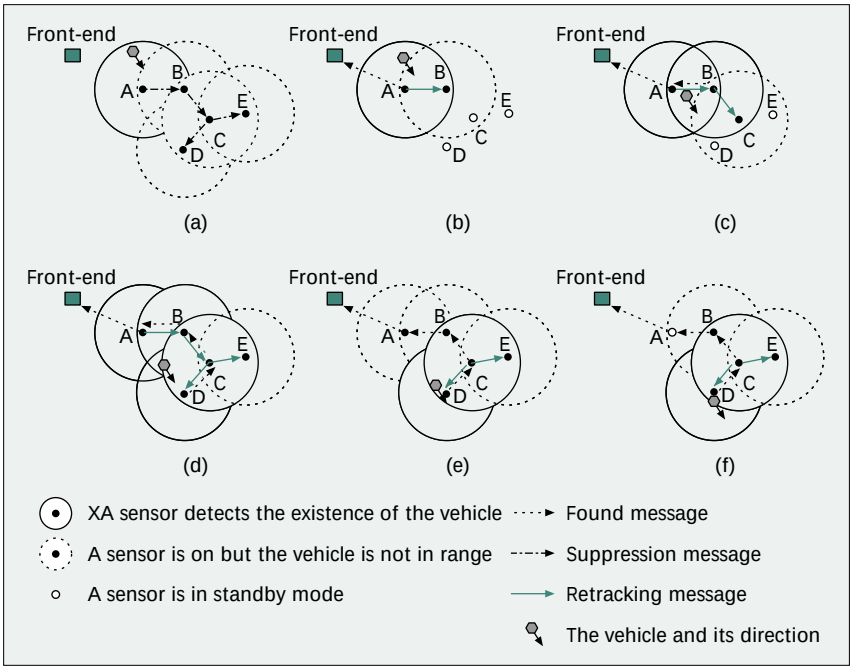

Figure 7. *a) A detects the incoming vehicle; b) the sensing activities of C, D, and E are suppressed but B starts tracking again; c) the vehicle comes into B's area and C restarts its sensor; d) C and D detect the vehicle and E's sensor is restarted; e) the vehicle goes out of A and B's ranges; f) sensing activity at A stops.*

retracking message, it restarts its sensing capability. In the retracking state, if a sensor node cannot detect the moving object for 40 s (the retracking interval), it stops sensing the object in order to conserve energy.

Results and Analysis — Table 3 presents the results obtained from both algorithms. Three metrics are of interest. First, we look at the efficiency of tracking and monitoring the moving object by measuring the ratio of *useful sensing* to the total amount of sensing. We defined useful sensing as sensing which successfully detects the vehicle. We found that the number of useful sensings from both algorithms are exactly the same (209) while the ordinary algorithm performed far more sensing activities (76,800 times compared to 8,828 times when using the coordinated algorithm). This is because of the lack of coordination among sensor nodes in the ordinary algorithm. Next, we counted the total number of packets sent out from all nodes for the entire simulation period. It is clear from the third column in Table 3 that the coordinated algorithm utilized more network bandwidth than the ordinary one. These extra packets accounted for all coordination-related packets (i.e., suppression and retracking messages). However, when we consider the total cost of operation, the coordinated algorithm is preferable, as shown in the last column. Here, we compare costs based on total cost of sensing (C_s) and transmitting (C_t) with ratio C_s:C_t = 4:1 [10], and then normalize the cost of the ordinary method to 1. The result shows that the coordinated method costs 17.9 percent of the ordinary method.

Figure 8 shows the result from another scenario, where the vehicle moves faster at 25 m/s. We varied the sensing intervals while keeping other parameters unchanged. From the figure, the number of useful sensings obtained from the coordinated algorithm is slightly lower than that obtained from the ordinary method when sensing intervals are lower than 15 s. The reason is that in the coordinated algorithm, we try to preserve network resources by suppressing sensing activity further away from the vehicle location and alerting only nodes nearby. Therefore, the number of sensor nodes monitoring the vehicle is far less than that of the ordinary method. However, at sensing intervals of 20 s and more, the coordinated algorithm hardly succeeded in detecting the moving vehicle. These results indicate that the reinitiation process of the coordinated algorithm could not keep up with the high mobility of the vehicle and the long sensing interval.

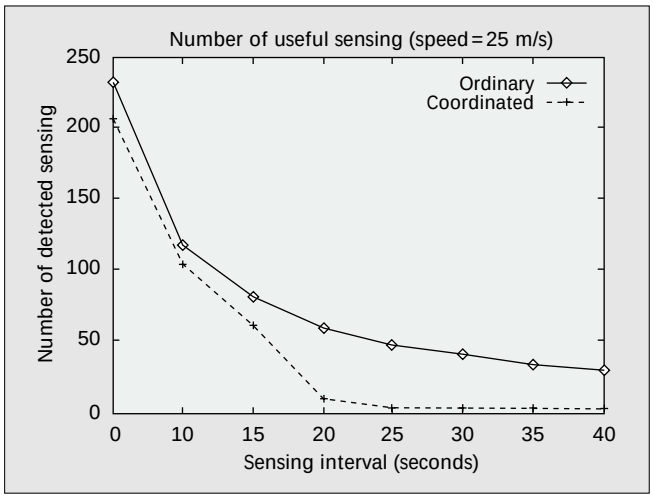

Figure 8. *Number of useful sensing from both methods when speed of the vehicle increases to 25 m/s.*

Summary

The advent of technology has facilitated the development of networked systems of small, low-power devices that combine programmable computing with multiple sensing and wireless communication capability. Soon our physical environment will be embedded with sensor nodes that enable new information gathering and processing capability. The sheer number of sensor nodes and the dynamics of their operating environments pose unique challenges in how information collected by and stored within the sensor network can be queried and accessed, and how concurrent sensing tasks can be executed internally and programmed by external clients. This article describes the SINA sensor information networking architecture that serves the role of middleware to facilitate querying, monitoring, and tasking of sensor networks. By integrating hierarchical clustering of sensor nodes and an attribute-based naming mechanism based on associative broadcast, SINA presents the associative spreadsheet abstraction that allows information to be organized and accessed according to specific application needs. The SINA kernel, represented by the collection of SEEs, implements three communication paradigms — sampling, self-orchestrated, and diffused computation operations — to facilitate information gathering and dissemination. On top the SINA kernel is a programmable substrate facilitated by the SQTL language to program sensing tasks. Sensor network querying and tasking applications are also presented together with their simulation studies.

References

[1] D. Estrin, R. Govindan, and J. Heidemann, "Embedding the Internet," *Commun. ACM*, vol. 43, May, 2000, pp. 38–41.
[2] T. Imielinski and S. Goel, "World," *IEEE Pers. Commun.*, vol. 7, Oct. 2000, pp. 4–9.
[3] A. Ward, A. Jones, and A. Hopper, "A New Location Technique for the Active Office," *IEEE Pers. Commun.*, vol. 4, Oct. 1997, pp. 42–47.
[4] B. Bayerdorffer, "Distributed Programming with Associative Broadcast," *Proc. 28th Hawaii Int'l. Conf. Sys. Sci.*, Jan. 1995.
[5] C. Jaikaeo, C. Srisathapornphat, and C.-C. Shen, "Querying and Tasking in Sensor Networks," *SPIE's 14th Annual Int'l. Symp. Aerospace/Defense Sensing, Simulation, and Control*, Orlando, FL, Apr. 2000.
[6] P. Bonnet, J. Gehrke and P. Seshadri, "Querying the Physical World," *IEEE Pers. Commun.*, vol. 7, Oct. 2000, pp. 10–15.
[7] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks," *Mobile Computing*, Kluwer, 1996, pp. 153–81.
[8] W. R. Heinzelman, J. Kulik, and H. Balakrishnan, "Adaptive Protocols for Information Dissemination in Wireless Sensor Networks," *ACM MobiComm '99*, Seattle, WA, Aug. 1999, pp. 174–85.
[9] C. Jaikaeo, C. Srisathapornphat, and C.-C. Shen, "Diagnosis of Sensor Networks," IEEE ICC, Helsinki, Finland, June 11–14, 2001.
[10] J. Byers and G. Nasser, "Utility-Based Decision-Making in Wireless Sensor Networks," Tech. rep. BU-CS 2000-014, Comp. Sci. Dept., Boston Univ., June 2000.

Biographies

CHIEN-CHUNG SHEN [M] (cshen@cis.udel.edu) received his B.S. and M.S. degrees from National Chiao Tung University, Taiwan, and Ph.D. from UCLA, all in computer science. He is an assistant professor in the Department of Computer and Information Sciences at the University of Delaware. His research interests include ad hoc and sensor networks, control and management of broadband networks, distributed object computing, and simulation.

CHAVALIT SRISATHAPORNPHAT (srisatha@cis.udel.edu) received his M.S. degree in computer and information sciences from the University of Delaware in 1999, and M.S. degree in computer sciences from Chulalongkorn University, Bangkok, Thailand, in 1995. He is currently a Ph.D. student in the Computer and Information Sciences Department at the University of Delaware. His research interests include ad hoc and sensor networks, power control, and distributed object computing.

CHAIPORN JAIKAEO (jaikaeo@cis.udel.edu) received his M.S. degree in computer and information sciences from the University of Delaware in 1999 and his B.Eng. degree in computer engineering from Kasetsart University, Bangkok, Thailand, in 1996. He is currently a Ph.D. student in the Computer and Information Sciences Department at the University of Delaware. His research interests include ad hoc and sensor networks, routing, and peer-to-peer computing.