Querying and Tasking in Sensor Networks

Chaiporn Jaikaeo Chavalit Srisathapornphat Chien-Chung Shen

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, U.S.A.

ABSTRACT

With the advancement of hardware technology, it becomes feasible to develop a networked system of pervasive computing platforms that combine programmable general purpose computers with multiple sensing and wireless communication capability. This networked system of programmable sensor nodes, together called a sensor network, poses unique challenges on how information collected by and stored within the sensor network should be queried and accessed, and how concurrent sensing tasks should be programmed from external clients. In this paper, we describe an architecture that facilitates querying and tasking of sensor networks. The key idea to the architecture lies in the development of the *Sensor Querying and Tasking Language* (SQTL) and the corresponding Sensor Execution Environment (SEE). We model a sensor network as a distributed set of collaborating nodes that carry out querying and tasking activities programmed in SQTL. A frontend node injects a message, that encapsulates an SQTL program, into a sensor node and starts a diffusion computation. A sensor node may diffuse the encapsulated SQTL program to other nodes as dictated by its logic and collaborately perform the specified querying or tasking activity. We will present the SQTL language and demonstrate its applicability using a maximum temperature querying application and a vehicle tracking application.

Keywords: querying and tasking language, sensor execution environment, sensor networks, SQTL

1. INTRODUCTION

The advent of hardware technology has facilitated the development of extremely small, low power sensor nodes that combine programmable general purpose computing with multiple sensing and wireless communication capability. Composing these sensor nodes into ad hoc computational and communication infrastructures to form sensor networks will enable new applications ranging from military situation awareness to factory process control. For instance, consider the following application scenario. Several thousand sensor nodes are rapidly deployed, for instance thrown from an airplane, in a disaster area. The sensor nodes communicate and coordinate to form an ad hoc communication network. Emergency response teams can emanate concurrent queries into the sensor network to collect environmental information in the disaster area. The queries are automatically routed to the most appropriate sensors, and replies are collected and fused en route to the designated reporting points. In addition, the sensor network may be commissioned to perform monitoring and tracking tasks in the disaster area to alert emergency response teams with changing situations of the physical environment. However, the sheer number of sensor nodes and the dynamics of their operating environments (for instance, limited battery power and hostile physical environment) pose unique challenges in the design of these operations.

Operations applied to a sensor network can be classified into two categories: querying and tasking. In our sensor network, we assume that there is one (or more) special entity, called *Frontend*, which acts as a gateway allowing users to interact with the sensor nodes. In order to acquire required information from those sensor nodes, a user specifies to the frontend node what information is needed. The frontend generates a formal statement, called a *query*, that will then be injected into the network. Once one or more replies are sent back, the frontend collects and processes these replies before handing the final result to the user. This process is called *querying*. Examples of querying a sensor network are as follows:

Further author information: (Send correspondence to Chien-Chung Shen)
Chaiporn Jaikaeo: jaikaeo@cis.udel.edu
Chavalit Srisathapornphat: srisatha@cis.udel.edu
Chien-Chung Shen: cshen@cis.udel.edu

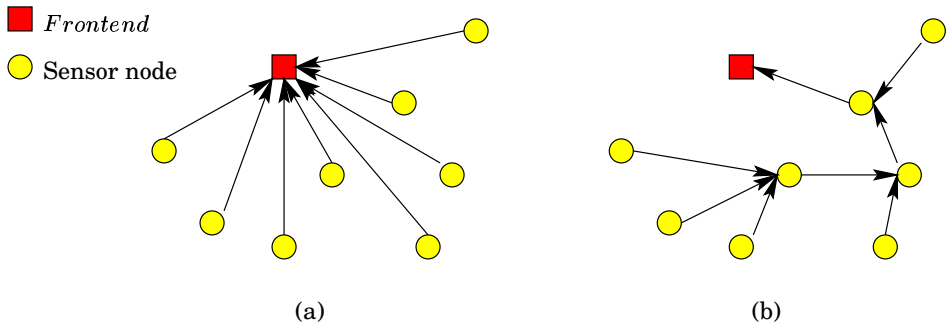

Figure 1. (a) All nodes reply to the frontend node, and (b) Data aggregation is done at some intermediate nodes

- what is the remaining power of node X?
- finding the current maximum temperature over a specific area.

From the above, querying can be considered as a synchronous operation in the sense that the frontend is blocked until the answer is received.

In contrast, some operations are impossible to be carried out by simple querying. User may want the sensor nodes to keep track of the environment changes, e.g. tracking moving vehicles. These activities require coordination among sensor nodes and might last for hours, days, or even longer. Assigning sensor nodes to execute such asynchronous operations is called *tasking*.

In research efforts such as Dataspace[1] and Device Database System,[2] sensor networks are modeled as distributed databases. Information retrieval is done via SQL[3]-like query languages. However, due to the fact that SQL is a declarative language, it lacks of operations to perform coordination tasks among sensor nodes. Although research in Device Database System[2] has extended SQL to support monitoring tasks, which they refer to as long-running queries, the enhancement still does not allow us to specify coordination between sensor nodes in detail. In this case, data flowing pattern depends on the internal database processing mechanism. The simplest way is to let the frontend node collect information from every sensor node and be responsible for processing all data, as illustrated in Fig. 1(a). However, this is impractical for the following reasons. Reply implosion at the frontend becomes a bottle neck and affects the overall performance when the number of nodes increases. Moreover, most of the time sensor nodes rely on wireless communication in order to interact with each other. Some nodes located far away from the frontend node may not be able to communicate to the frontend directly due to limited transmission power. Users have no freedom to modify the interaction between nodes based on their preferences when they are allowed to use only the given query language.

In this paper, we describe an architecture that facilitates querying and tasking of sensor networks. The key idea to the architecture lies in the development of the *Sensor Querying and Tasking Language* (SQTL) and the corresponding *Sensor Execution Environment* (*SEE*). Instead of modeling a sensor network as a distributed database where all sensor nodes are passive, we model a sensor network as a distributed set of collaborating nodes with active, programmable capability, which allows nodes to coordinate with each other in order to achieve an assigned task. The sensor nodes then become active and autonomous. Nodes can be programmed to have more interaction so that user's queries can be carried out more efficiently. For example, Fig 1(b) shows how data aggregation can be done via interaction between nodes instead of being done solely at the frontend node.

In our work, we have developed the SQTL scripting language, which is easy for users to program and for sensor nodes to interpret, and supports both querying and tasking operations. The language provides programmers, or even end users, flexibility to determine behavior of the sensor nodes according to the programmers' preferences. SQTL also provides handy data structures and primitives, and is designed to be light-weight and object-oriented including event handling constructs, which make it easy to program.

The remainder of the paper is organized as follows. The SQTL language and the SEE execution environment are described in the next section. Querying and tasking applications of using SQTL are described in section 3. Section 4 concludes the paper and describes future research work.

2. SENSOR QUERY AND TASKING LANGUAGE (SQTL)

In order for sensor nodes to perform a query or a task collaboratively and autonomously, they must be provided with clearly specified algorithms (or programs) for executing the query or task. We have developed the Sensor Querying and Tasking Language (SQTL) to achieve these goals. Before an SQTL program is injected into a sensor network, it will be encapsulated in a wrapper called *SQTL wrapper*. We choose to modify KQML performatives (as described by Labrou and Finin[4]) and use them as *actions* of SQTL wrapper for SQTL programs because of their simplicity and strong expressive power. SQTL programs together with SQTL wrappers are interpreted and executed by the *Sensor Execution Environment* (*SEE*) located in each sensor node. The next subsection describes features of SQTL. How an SQTL script is encapsulated and executed at a sensor node is described in subsection 2.2. Subsection 2.3 describes the language constructs of the SQTL language. Primities provided by SEE are described in subsection 2.4.

2.1. SQTL Features

SQTL includes capabilities of both object oriented and procedural languages. Most primitives provided by the Sensor Execution Environment as a part of the SQTL language are in the form of classes. These classes can be instantiated by application programs (SQTL programs generated at the *Frontend* to satisfy user requirements can be called application programs), which allows accesses to most system resources, e.g. individual sensing device or a group of the same functionality sensors in a sensor node. Note that hiding these physical sensor devices from being directly accessed by applications allows the underlying operating system to transparently aggregate results from more than one sensor of each particular sensor type. For example, a sensor node may contains more than one motion detection sensor. Normally, those sensors are directed in different directions to maximize sensing area. An instantiation of this motion sensor class will result in a logical object that represents a group of all sensors in this class. Results from every sensor can be summarized and reported to the calling application when the application invoke a specific primitive to retrieve the result. However, the language does not provide inheritance of system classes neither creation of user-defined classes.

Messages from other nodes and sensing results from instantiated logical devices are received through an event handling mechanism, which is a part of the language. The results from sensor devices are also manipulated by a program written in a classical procedural style. The final result will then be delivered to the *Frontend*.

2.2. SQTL Wrapper and Execution Environment

As mention above, a complete SQTL program within an SQTL wrapper generated by the *Frontend* will be distributed into one or more sensor nodes. A collection of specific actions are defined to suit the requirements in passing these programs between *Frontend* and sensor nodes and among sensor nodes themselves. Some of those are described in Table 1. *Actions* also require parameters, which are called *action parameters*. Most of them are in similar meaning as defined by Labrou and Finin,[4] while some additional parameters have been added or modified to suit our code passing scheme between nodes. We summarized parameters and their semantics in Table 2.

At the sensor node, an SEE, which has the capability of dispatching incoming messages, examines all arrival SQTL messages and performs the appropriate operation for each type of *action* specified in the messages. SEE always looks inside the *:receiver* parameter and makes a decision based on value in this field regarding where to forward each message.

Messages with "ALL_NODES" in their *:group* subparameters, will be re-broadcast to every sensor nodes in the network and those with "NEIGHBORS" will be forwarded to only the nodes' one-hop-away neighbors. For other messages, which have specific receiver node identifiers in *:group*, will be received only at that target node. Furthermore, a sender also has a choice to specify the receiver's characteristic in the *:criteria* subparameter if the sender does not know exactly which nodes will be the receivers at the sending time. One way to specify criteria is to use attribute-value pairs. With the use of *:criteria*, binding between criteria and actual receiver's

Table 1. Actions used in SQTL wrapper

Action	Meaning
tell	sender sends information to receiver(s)
install	sender sends a program to install into receiver's memory
start	a command from sender to start a pre-installed program at a receiver
execute	a combination of install-start-uninstall actions
uninstall	sender asks receiver to remove a pre-installed program
stop	sender asks receiver to completely stop running an application
suspend	sender asks receiver to suspend a currently running application
resume	sender asks receiver to resume the execution of a suspended application
flush	sender asks receiver to terminate an executing SQTL program and immediately erases it from the receiver's memory

Table 2. Parameters used by Actions in SQTL wrapper

Parameter	Meaning
:sender	the sender of an SQTL message wrapper
:receiver	potential receivers specify by two following subparameters
:group	subparameter of :receiver to specify group of receiver
:criteria	subparameter of :receiver to specify selection criteria of receivers
:application-id	the unique id for each application in the same sensor network
:language	specify a language used in :content
:content	a payload containing a program, a message or return values
:with (optional)	tuples of parameters used in the program passed from sender to receiver

characteristics will be made by SEE of the receiver node. SEE only accepts messages if the node can satisfy *:criteria*. This process is called *late binding* and has been described earlier as one feature of Associative Broadcast by Bayerdorffer.[5]

All messages with a known *application-id* will also be processed locally. In this case, a message with *tell* action will be delivered to the corresponding application and will be further processed by the application. All other actions will be processed by SEE. A message of type *install* will cause SEE to store the SQTL code specified in the *:content* part of the message into sensor's storage area with an index specified in the *:application-id* parameter without any further operation. The code quietly stays and waits in the memory until SEE receives an SQTL message with action *start*, then SEE will start executing this code. The duration of execution can be both pre-determined inside the SQTL code or on-line controlled by other messages of action *stop* or *suspend*. If the application is suspended, it can continue from the last stopping point when receives a *resume* message. The other two actions, *uninstall* and *flush*, cause SEE of the sensor node who receives these messages to remove the installed code. An SEE that receives *uninstall* messages waits until the currently running application completes before it removes the code from the node storage space. On the other hand, receiving a *flush* message causes SEE to cease and remove the specified SQTL application program immediately even it is in the middle of execution. Using this code storing and executing approach, the bandwidth spent in disseminating the same SQTL programs when they are needed can be substantially reduced, given that a sensor node is equipped with enough memory space and the program size is small.

In practice, some programs that serve only ad hoc querying or tasking will be executed only once. The execution of these disposable codes should be invoked through the *execute* action. Notice that SQTL program is designed for sensor networks with underlying active technology.[6] Once an SQTL code is injected from the

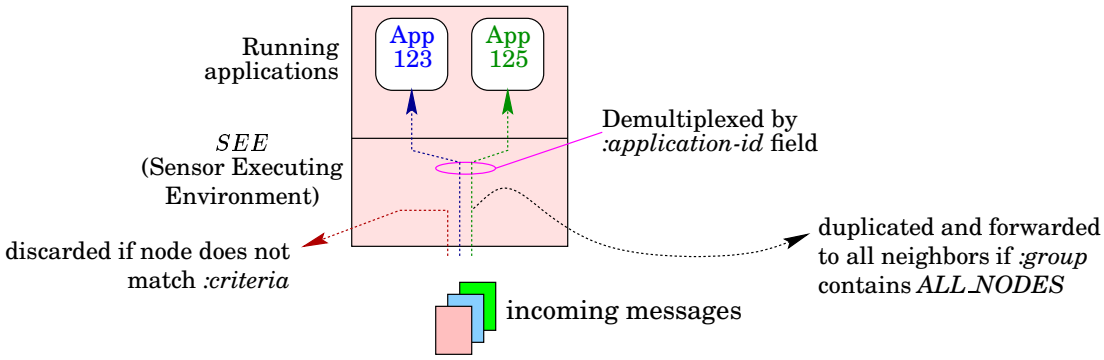

Figure 2. Dispatching of messages received by a sensor node

Frontend to one or more sensor node, the code can be pushed to other sensors in order to complete a task assinged inside the code. After a result is produced at each individual sensor node, a *tell* message will be generated by the SQTL application to deliver the result back to the targets of that result (normally be the upstream sensor node where the code came from or they can be any other node specifies in the code given that lower layer routing protocols are provided). Notice also that all actions are targeted at SEE, i.e. SEE does not pass these SQTL messages up to the running applications. The only exception is for an *action tell*, which will be checked to see if it requires any re-forwarding by *SEE*. After finishing the re-forwarding process, it will be delivered to the specified *:application-id*. Fig. 2 depicts the dispatching of incoming messages performed by the SEE.

One feature of the SQTL wrapper is its flexibility to allow program written in other languages to be carried by SQTL message wrapper. For example, the *Frontend* may specify SQL as a value of *:language* parameter and embedded an SQL statement in the *:content* parameter. However, at the sensor node, it must have an SQL engine implemented in order to be able to execute this embedded SQL statement. The embedded SQL statement will be delivered to the SQL engine by SEE.

Additional to demultiplexing incoming SQTL messages, SEE takes care of outgoing SQTL messages from all running applications as well. Outgoing messages will be distributed to target(s) specified in *:receiver* parameter through the underlying communication mechanism.

2.3. SQTL Language Constructs

As mention earlier, SQTL is designed to be similar to a procedural language with light-weight object-oriented feature. The language constructs include: arithmetic (`+`, `-`, `*`, `/`), comparison (`==`, `!=`, `<`, `>`), and boolean (**AND, OR, NOT**) operators, assignments, conditional construct (**if-then-else**), loop construct (**while**), object instantiation (**new**), and event handling construct (**upon**). There is no variable declaration block which means variables can be created on demand and can be of any type just like the language feature provided by Tcl.[7] Most language constructs described above are used in the same way as other procedural languages.

For many sensor network applications, sensor nodes are often 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, there are three types of events supported by SQTL: (1) events generated when a message is received by a sensor node, (2) events triggered periodically by a timer, and (3) events caused by timeout. These types of events are defined by the SQTL keywords **receive**, **every**, and **timeout**, respectively.

2.4. SEE-provided Primitives

There are a number of primitives provided by the Sensor Execution Environment. They can be categorized into groups according to their functionality as follows:

- **sensor access primitives**, e.g. getTemperatureSensor(), turnOn(), turnOff(),

- **communication primitives**, e.g. tell(), execute(), send(), and
- **location-aware primitives**, e.g. isNorthOf(), isNear(), isNeighbor().

In addition to system-provided primitives listed above, SQTL also provides classes implementing basic data structures such as array and linked-list. Data aggregate functions, e.g. maximum, minimum, and average, can be applied to the data structures as well.

3. SAMPLE APPLICATIONS

In this section, we describes two example applications of using SQTL scripts to query and task a sensor network. For querying, a distributed algorithm to find the maximum temperature in the area monitored by the sensor network is presented in subsection 3.1. In subsection 3.2, tasking over a sensor network is demonstrated through a coordinated algorithm for tracking a moving vehicle. The two algorithms are presented with the following assumptions.

- The sensor network is not partitioned.
- All sensor nodes are homogeneous in their capabilities.
- There is no packet lost, no communication conflict, and all communications are symmetric.
- No sensor nodes fail during the time the algorithms are being executed.
- By nature of communication in wireless sensor networks, all communications between sensor nodes are broadcasting. Sender nodes can specify a particular receiver by using attribute matching.
- The network is not expected to have routing support provided from lower layers. However, applications are able to keep track of the sender's address and use reverse path forwarding to deliver results back to the sender.

3.1. Maximum Temperature Finding

This first example illustrates how SQTL can be used to program sensor nodes to carry out a query with data aggregation. Assuming that all sensor nodes in the network are capable of sensing temperature, we are to find out what the highest temperature is. With the flexibility of SQTL, and the ability of installing and executing mobile code,[8] a specific SQL engine would be downloaded into the sensor nodes so that an SQL statement, encapsulated inside an SQTL wrapper as shown below, could be interpreted at a sensor node.

```
(execute
   :sender          FRONTEND
   :receiver        (:group NODE(1) :criteria TRUE)
   :application-id  123
   :language        SQL
   :content         ( SELECT Max(getTemperature()) FROM ALL_NODES ))
```

However, it is not trivial to design an SQL engine to be both generic and efficient at the same time. Although generality could be achieved, the engine might end up with a centralized design which results in the implosion problem as describe in Section 1. In this case, more task-descriptive code might be preferable to one single SQL statement. Fig. 3 presents a distributed version of the maximum temperature query written in SQTL. Sensor nodes are programmed to have more interaction between each other, instead of overflowing a particular node.

To show how the algorithm works, let us consider Fig. 4. When the program starts running, the frontend node picks a nearby node to deliver the *Findmax* code. Let us assume that the node is node A. Once node A receives and executes the code, it confirms the *Frontend* by sending a 'confirmation' message. Node A then broadcasts the *Findmax* code to all of its neighbors, B, C, and D, which will be recursively executing the same code and send confirmation messages back to Node A. These steps are presented in Fig. 4(a), 4(b) and 4(c),

```
(execute
   :sender          FRONTEND
   :receiver        (:group NODE(1) :criteria TRUE)
   :application-id  123
   :language        SQTL
   :content
   (
      tell(MESSAGE.sender, "TRUE", "confirm");
      execute(NEIGHBORS, "TRUE", MESSAGE.content);
      confirmCount = 0;   // count the number of responded confirm messages
      upon {
         receive(msg) where msg.content == "confirm" :
            confirmCount++;
         timeout(500) :
            break;
      }

      // create a list of returned temperature values
      answerList = new List();

      // add my own temperature into the list
      answerList.add(getTempSensor().getCurrentTemp());

      upon {
         receive(msg) where msg.action == "return": {
            answerList.add(msg.content);
            confirmCount--;
            if (confirmCount == 0) break;
         }
      }

      // return the maximum back to the sender
      tell(MESSAGE.sender, "TRUE", answerList.max());
   ))
```

Figure 3. SQTL code with SQTL wrapper for the distributed maximum temperature finding algorithm

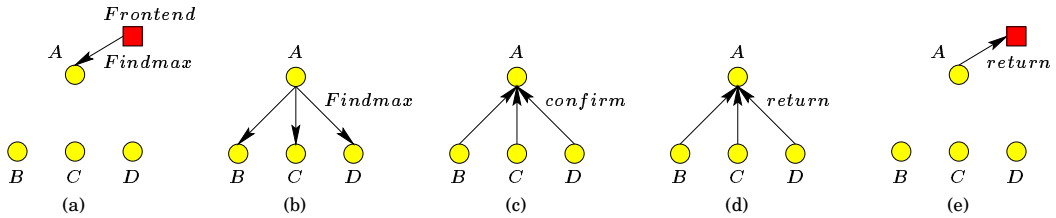

Figure 4. (a) The frontend node injects the *Findmax* code into Node A, (b) Node A broadcasts the *Findmax* code, (c) A's children send back confirmation messages, (d) A's children return local highest temperatures, and (e) A compares and returns the highest temperature to the frontend node

respectively. As shown in the code from Fig. 3, Node A will keep track of the number of received confirmation messages until it times out. Note that all messages passed between nodes are encapsulated inside the SQTL wrappers.

In Fig. 4(d), each of A's children is sending back a return packet which contains the highest temperature of all the nodes underneath it (not shown). After collecting the returned messages from all of its children, Node A then compares the temperature values extracted from the messages with its own value and calculates the maximum. This value is returned to the frontend node and the execution of the algorithm terminates, as presented in Fig. 4(e).

3.2. Coordinated Vehicle Tracking

To illustrate the capability of SQTL in tasking a sensor network, we use a coordinated version of vehicle tracking as an example. The main purpose of vehicle tracking in a sensor network is to locate the specified vehicle and monitor its movement. One way to track a moving vehicle is to let all sensors in a network execute the tracking applications independently without any coordination. Sensors that notice the existence of the vehicle deliver tracked vehicle information back to the *Frontend* . Although this method works well regardless of where the vehicle is in the area, all sensors have to spend energy and processing power to detect the appearance of the vehicle. In contrast, we have developed an algorithm in SQTL that takes advantage of coordination among sensor nodes to efficiently track moving vehicles and, at the same time, preserve scarce network resources. In this algorithm, we have the following three additional assumptions. First, a vehicle has an active tag attached, which can be detected by sensors. Second, the sensing range of sensors is equal to sensors' transmission range. Third, a target vehicle may or may not be in the network sensing range when the application starts, i.e. it may come into the sensor network region at a later time.

Fig. 5 presents the algorithm as written in SQTL. We describe the algorithm by using an example in Fig. 6. When the *Frontend* sends an SQTL message to one node in the network, it disseminates this message through out the network. Eventually, all nodes in the network start their motion sensing activities. When the vehicle comes into the sensing area, node A is the first node noticing the appearance of the vehicle. A will then broadcast 'suppression' messages to the whole network (Fig. 6(a)). Other nodes will set their motion sensors to the standby mode when receiving the message. After doing this, A also broadcasts a 're-tracking' message to only its one-hop neighbors, which include only B in this case. The 're-tracking' message causes B to restart its sensor again. A then sends a reply back to where it received the SQTL program from, which is the *frontend* (Fig. 6(b)). When the vehicle moves into B's sensing area, B also have to broadcast a 're-tracking' message to only its one-hop neighbors, and send a reply back to its sender (which is A). Now, C restarts its motion sensor after received the message from B (Fig. 6(c)). In Fig. 6(d), the vehicle moves into the proximity of C and D, and the same process happens at C and D as it occurred at A and B. In Fig. 6(e), even though the vehicle move out of A's and B's range, they still continue tracking for a certain period of time, and have to forward all replies from C and D back to the *frontend* as well. In Fig. 6(f), A's retracking time is expired and it stops sensing the vehicle, but B still receives a 're-tracking' message from C, so that B will continue its sensing activity.

As clearly seen from this sample application, tasking a sensor network to collaborately and autonomously accomplish one specific mission could be performed through the event driven capability of SQTL, which might not be easily done by approaches based on SQL alone.

4. CONCLUSION AND FUTURE WORK

In few years, our physical environment will be embedded with billions of 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 on how information collected by and stored within the sensor network should be queried and accessed, and how concurrent sensing tasks should be programmed from external clients. In this paper, we describe the Sensor Querying and Tasking Language and the corresponding Sensor Execution Environment that facilitate querying and tasking of sensor networks. Queries and tasks are programmed in SQTL and injected into the sensor network. Sensor Execution Environments located in sensor nodes interpret the SQTL program and collaboratively execute the specified operations. The applicability of SQTL for querying and tasking operations are illustrated by the maximum temperature finding and the moving vehicle tracking applications.

Research is in progress to investigate issues involving interactions between mobile sensors and fixed sensor networks. For instance, a mobile sensor issues a query into the fixed sensor network while marching ahead (Fig. 7(A)). When the reply becomes available, the mobile sensor's point of contact with the fixed sensor network has changed. We are studying two reply forwarding schemes. The first scheme requires the mobile sensor to update its current point of contact with the sensor where the query was issued (Fig. 7(B)), while the second scheme requires the sensor to which the query was issued to actively seek out the current location of the mobile sensor.

ACKNOWLEDGEMENTS

This paper is based upon work supported in part by a grant from Telcordia Technologies. The authors would like to thank Drs. Deh-phone Hsing and Yukun Tsai for their comments and support.

REFERENCES

1. T. Imieliński and S. Goel, "Dataspace - querying and monitoring deeply networked collections of physical objects," in *Proceedings of International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE'99)*, (Seattle, Washington), August 1999.
2. J. G. Philippe Bonnet and P. Seshadri, "Query Processing in a Device Database System." http://www.cs.cornell.edu/database/cougar/cougar.pdf, March 2000.
3. S. Vang, *SQL and Relational Databases*, Microtrend Books, San Marcos, CA, 1991.
4. Y. Labrou and T. Finin, "A Proposal for a New KQML Specification," tech. rep., Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997.
5. B. Bayerdorffer, "Distributed Programming with Associative Broadcast," in *Proceedings of the Twenty-eighth Hawaii International Conferenc on System Sciences*, January 1999.
6. D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine* **35**, January 1997.
7. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
8. A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Transactions on Software Engineering* **24**, May 1998.

```
(execute
   :sender          Frontend
   :receiver        (:group NODE[0] :criteria TRUE)
   :application-id  320984
   :language        SQTL
   :with
   (  clocktype   trackingTime      500,              // application running time
      clocktype   reTrackingTime    50,               // retracking interval
      clocktype   trackingFrequency 1,                // tracking frequency
      object      target            Vehicle1)         // target vehicle
   :content
   (  timerApplication = new Timer(trackingTime);     // instantiate a timer
      timerApplication.start();                       // turn it on
      timerReTracking  = new Timer(reTrackingTime);
      timerReTracking.reset();                        // reset the timer to expire
      execute (MESSAGE.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) where (sensor1.detect(target)): {
               tell (MESSAGE.ALL_NODES, "TRUE", "suppress");
               tell (MESSAGE.NEIGHBORS, "TRUE", "retrack");
               tell (MESSAGE.sender, "TRUE",    "found");
               timerReTracking.start();
               break;
            }
            expire (timerApplication): sensor1.turnOff(); exit(1);
         }
         // After one sensor node sees the vehicle
         upon {
            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 (sender, "TRUE", "found");
            every (trackingFrequency) where (sensor1.detect(target)): {
               tell (MESSAGE.sender, "TRUE",    "found");
               tell (MESSAGE.NEIGHBORS, "TRUE", "suppress");
               tell (MESSAGE.NEIGHBORS, "TRUE", "retrack");
               timerReTracking.start();
            }
            expire (timerReTracking) : sensor1.standby();
            expire (timerApplication): sensor1.turnOff(); exit(1);
         }
      }
      else
         exit(1);
   ) // end content
) // end execute
```

Figure 5. SQTL code with SQTL wrapper for the coordinated vehicle tracking algorithm

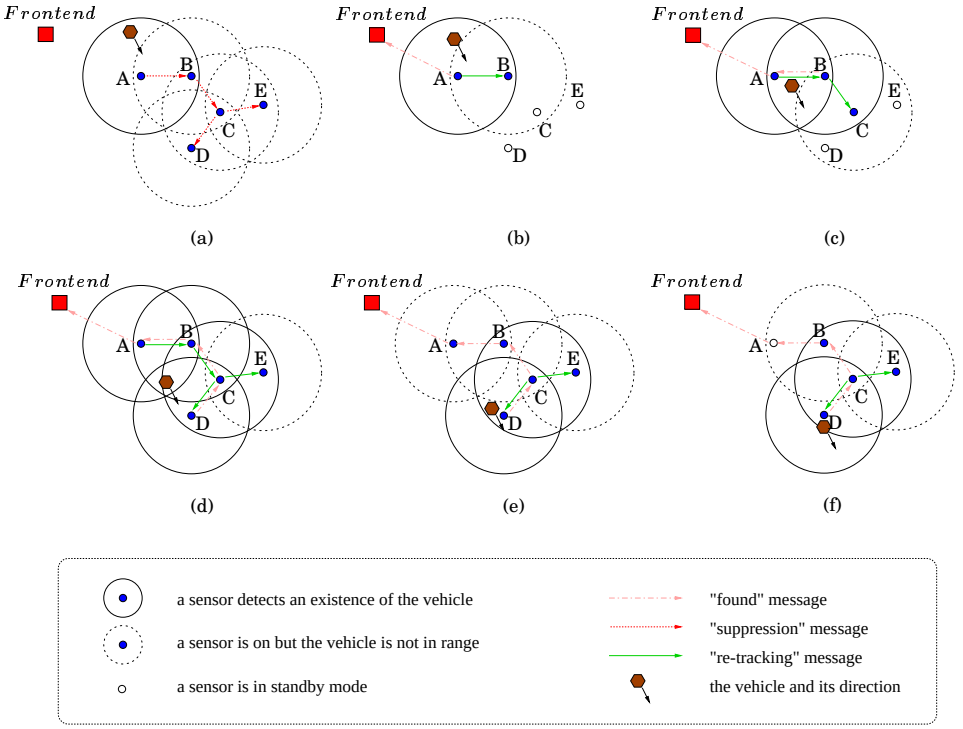

Figure 6. (a) A detects the incoming vehicle, (b) the sensing activities of C, D and E are suppressed but B starts tracking again, (c) the vehicle comes in to B's area and C restarts its sensor, (d) C and D detect the vehicle and E's sensor is restarted, (e) the vehicle goes out of A and B's ranges, and (f) sensing activity at A stops

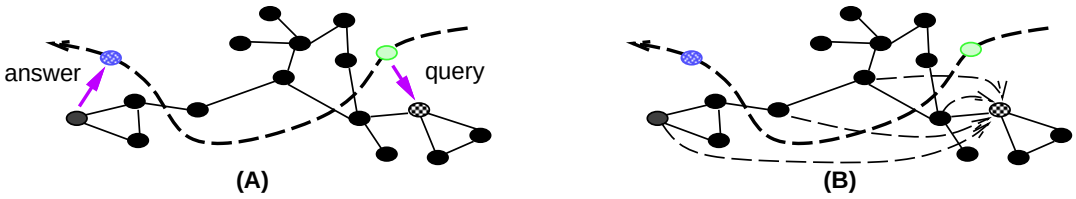

Figure 7. Interworking between mobile and fixed sensor nodes