Enabling Local Communication for Immersive Mobile Computing

Sanem Kabadayi
Christine Julien

TR-UTEDGE-2006-004

A Local Data Abstraction and Communication Paradigm for Pervasive Computing

Sanem Kabadayı and Christine Julien
The Center for Excellence in Distributed Global Environments
The Department of Electrical and Computer Engineering
The University of Texas at Austin
{s.kabadayi, c.julien}@mail.utexas.edu

Abstract

As sensor networks are increasingly used to support pervasive computing, we envision an instrumented environment that can provide varying amounts of information to mobile applications immersed within the network. Such a scenario deviates from existing deployments of sensor networks which are often highly application-specific and funnel information to a central collection point. We instead target scenarios in which multiple mobile applications will leverage sensor network nodes opportunistically and unpredictably. Such situations require new communication abstractions that enable immersed devices to interact directly with available sensors, reducing both communication overhead and data latency. This paper introduces scenes*, which applications create based on their communication requirements, abstract properties of the underlying network communication, and properties of the physical environment. This paper reports on the communication model, an initial implementation, and its performance in varying scenarios.*

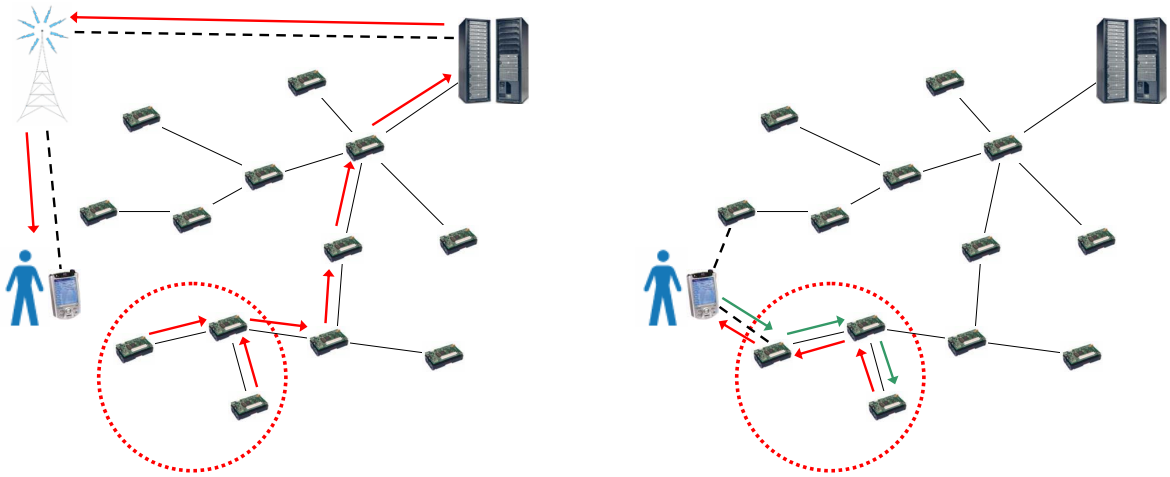
1. Introduction

Sensor networks, consisting of numerous miniature, battery-powered devices that communicate wirelessly to gather and share information, have emerged as an integral component of pervasive computing environments. Much of the existing work focuses on application-specific networks where the nodes are deployed for a particular task, and sensor data is collected at a central location to be processed and/or accessed via the Internet. We envision a more futuristic (but not unrealistic) scenario in which sensor networks are general-purpose and reusable in support of pervasive computing. While the networks may remain domain-specific, we target situations in which the applications that will be deployed are not known *a priori* and may include varying adaptive behaviors, for example in aware homes [16], intelligent construction sites [14], battlefield scenarios [9], and first responder deployments [19]. These applications involve users who access locally-sensed information on-demand, which is exactly the vision of pervasive computing [28], in which sensor networks must play an integral role [6].

In current pervasive computing applications, communication is often accomplished using mobile ad hoc network routing protocols that deliver messages using intermediate nodes as routers [25, 13, 26, 4, 2]. Because these protocols are tailored for providing routes across networks that can grow very large, they do not favor the local interactions common in pervasive computing. In addition, the protocols require senders and receivers to have explicit knowledge of each other. In pervasive computing, however, a client device often has no *a priori* knowledge about network components with which it will interact.

This paper introduces the scene abstraction and a network protocol that provides the abstraction for developers of pervasive computing applications. The scene abstraction allows an application's operating environment to include a dynamic set of devices embedded in the environment. As the device on which the application is running moves, the scene automatically updates to reflect changing conditions, thereby enabling access to local data.

We focus on supporting applications in which *client devices* (e.g., laptops or PDAs) interact directly with embedded sensor networks. While this style of interaction is common in many application domains, we will refer to applications from the first responder domain, which provides a unique and heterogeneous mix of embedded and mobile devices. The former includes fixed sensors in buildings and environments that are present regardless of crises and ad hoc deployments of sensors that responders may distribute when they arrive. Mobile devices include those moving within ve-



(a) Using today's capabilities, a user's query is physically detached from the sensor field. Query resolution relies on a centralized point, generally the *root* of a routing tree constructed over the network.

(b) Using scenes, an application on the user's device can seamlessly connect to the sensors in the local region, removing the requirement that physically distant sensors participate in routing.

Figure 1. Comparison of (a) existing operational environments with (b) the scene **operational environment**

hicles, carried by responders, and even autonomous robots that may perform exploration and reconnaissance.

The remainder of this paper is organized as follows. Section 2 carefully defines the problem and its underlying motivation. We informally characterize our scene abstraction in Section 3 and its implementation in Section 4. Section 5 demonstrates an example scene. Section 6 gives performance results for this implementation, Section 7 overviews related projects, and Section 8 concludes.

2. Problem Definition and Motivation

In pervasive computing applications, users move through instrumented environments and desire on-demand access to locally gathered information. Figure 1(a) shows how interactions with sensor networks commonly occur, while Figure 1(b) shows the needs of our intended applications. While existing collection behaviors that sense, aggregate, and stream information to a central collection point may be useful for other aspects of pervasive computing, this paper undertakes the portion of the problem related to enabling applications' direct, on-demand, and mobile interactions. This style of interaction differs from common uses of sensor networks, introducing several unique challenges and heightening existing ones:

- *Locality of interactions*: A client device interacts directly with embedded devices. While this can minimize the communication overhead and latency (see Figure 1), it can be difficult for an application to precisely specify the area from which it collects information.
- *Mobility-induced dynamics*: While embedded devices are likely stationary, the application interacting with them runs on a mobile device. Therefore, the device's connections to particular sensors and the area from which the application desires to draw information are subject to constant change.
- *Unpredictability of coordination*: Pervasive computing demands that networks be general-purpose (i.e., support applications whose natures are not known *a priori*). As such, few assumptions can be made about applications' needs or intentions, requiring networks to adapt.

The confluence of these challenges necessitates the development of a new paradigm of communication for ubiquitous computing applications that pays careful attention to the design issues described above.

3. Scenes: Declarative Local Interactions

In a pervasive computing network, the set of data sources near a user changes based on the user's movement. Furthermore, if the network is well-connected, the user's device will be able to reach vast amounts of raw information that must be filtered to be usable. The application must be able

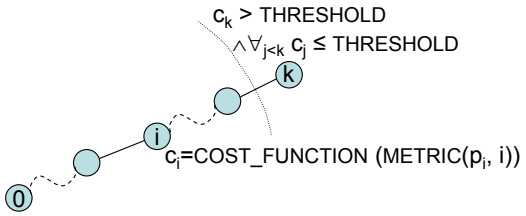


Figure 2. Distributed scene **computation**

to limit the scope of its interactions to include only the data that matches its needs. In our model, an application's operating environment (i.e., the sensors with which it interacts) is encapsulated in an abstraction called a scene that constrains which particular sensors influence the application. This abstraction allows local, multihop neighborhoods of possibly heterogeneous devices surrounding a particular application, supports mobility by dynamically updating the scene's participants, and minimizes how much the application developer must know about the underlying implementation. The constraints that define a scene may be on properties of hosts (e.g., battery life), of network links (e.g., bandwidth), and of data (e.g., type).

3.1. Defining Scenes

The declarative specification defining a scene allows an application programmer to flexibly describe the type of scene he wants to create. Multiple constraints can be used to define a single scene. The programmer only needs to specify three parameters to define a constraint:

- *Metric*: A property of the network or environment that defines the cost of a connection.
- *Path cost function*: A function (such as SUM, AVG, MIN, MAX) that operates on a network path.
- *Threshold*: The value a path's cost must satisfy for that sensor to be a member of the scene.

Thus, a scene, S, is specified by one or more constraints, $C_1, \ldots, C_n$:

$$C_1 = \langle M_1, F_1, T_1 \rangle, \ldots, C_n = \langle M_n, F_n, T_n \rangle$$

where M denotes a metric, F denotes a path cost function, and T denotes a threshold.

Figure 2 demonstrates the relationships between these components. This figure is a simplification that shows only a one-constraint scene and a single network path. The cost to a particular node in the path (e.g., node i) is calculated by applying the scene's path cost function to the metric. The latter can combine information about the path so far (p_i) and information about this node. Nodes along a path continue to be included in the scene until the path hits a node whose cost (e.g., c_k) is greater than the scene's threshold. This functionality is implemented in a dynamic distributed algorithm that can calculate (and dynamically recalculate) scene membership. The application's messages carry with them the metric, path cost function, and threshold, which suffice to enable each node to independently determine whether it is a member of the scene. Each node along a network path determines whether it lies within the scene, and if so, forwards the message. It is possible for a node to qualify to be within the scene based on multiple paths. If a node receives a scene message that it has already processed, and the new metric value is not shorter, the new message is dropped. If the new message carries a shorter metric, then the node forwards the information again because it may enable new nodes to be included in the scene.

The scene concept conveys a notion of locality, and each application decides how "local" its interactions need to be. A first responder team leader coordinating the team spread throughout the site may want to have an aggregate view of the smoke conditions over the entire site. On the other hand, a particular responder may want a scene that contains readings only from nearby sensors or sensors within his path of movement. The scene for the leader would be "all smoke sensors within the site boundaries", while the scene for the responder might be "all smoke sensors within 5m." As a responder moves through the site, the scene *specification* stays the same, but the data sources belonging to the scene may change.

To present the dynamic scene construct to the developer, we build a simple API that includes general-purpose metrics (e.g., hop count, distance, etc.) and provides a straightforward mechanism for inserting new metrics. Applications on client devices specify scenes through a Java programming interface, and these specifications are translated into the appropriate low-level code before queries are sent to the sensors. Figure 3 shows the Java API for the `Scene` class.

```
class Scene{
   public Scene(Constraints[] c);
   public void send(AppMessage a);
   public void maintain(AppMessage a);
}
```

Figure 3. The API for the `Scene` **class**

From the application's perspective, a scene is a dynamic data structure containing a set of qualified sensors which are determined by a list of constraints, `Constraints[]`, and accessed through the latter two methods: `send()` and `maintain()`. The `send()` method poses a one-time query to scene members to which each recipient sends at most

Table 1. Example scene definitions

	Hop Count Scene	Battery Power Scene	Distance Scene
Metric	SCENE_HOP_COUNT	SCENE_BATTERY_POWER	SCENE_DISTANCE
Cost Function	SCENE_SUM	SCENE_MIN	SCENE_DFORMULA
Metric Value	number of hops traversed	minimum battery power	location of source
Threshold	maximum number of hops	minimum allowable battery power	maximum physical distance

```
Scene s = new Scene({new Constraint(Scene.SCENE_DISTANCE, Scene.SCENE_DFORMULA,
                                                    new IntegerThreshold(5)},
                    {new Constraint(Scene.SCENE_LATENCY, Scene.SCENE_MAX,
                                                    new IntegerThreshold(15)} ) ;
```

Figure 4. An example `scene` declaration

one reply. This is similar to a multicast, but the significant difference is that the receivers are *dynamically* determined by the parameters defining the scene. The `maintain()` method sends a persistent query to the scene, implicitly requesting that the scene structure be maintained, even as the participants change.

Table 1 shows examples of how scenes may be specified. These examples include restricting the scene by the maximum number of hops allowed, the minimum allowable battery power on each participating node, or the maximum physical distance. As one example, SCENE_HOP_COUNT effectively assigns a value of one to each network link. Therefore, using the built-in SCENE_SUM path cost function, the application can build a hop count scene that sums the number of hops a message takes and only includes nodes that are within the number of hops as specified by the threshold. The scene can be further restricted using latency as a second constraint. For example, in a first responder deployment, the code shown in Figure 4 defines a scene that includes every sensor within 5m of the declaring device and has latency less than 15ms. It is possible to extend the number of constraints inductively, and this combinatorial nature provides significant flexibility to the programmer.

3.2. Maintaining Scenes

While a scene provides the appearance of a dynamic data structure, the implementation behaves on-demand. If the application is not actively messaging the sensors, no proactive behavior occurs. Only when the application uses a scene does the protocol communicate with other local devices.

For one-time queries, a scene is created, and, once the response to the query has been obtained from this scene, the scene is not stored or updated in any way. On the other hand, if the scene is to be used for a persistent query, the scene needs to be maintained. To maintain the scene, each member sends periodic beacons advertising its current value for the metric. Each node also monitors beacons from its parent in the routing tree, whose identity is provided as previous hop information in the original scene message. If a node has not heard from its parent for three consecutive beacon intervals, it disqualifies itself from the scene. This corresponds to the node falling outside of the span of the scene due to client mobility or other dynamics. In addition, if the client's motion necessitates a new node to suddenly become a member of the scene, this new node becomes aware of this condition through the beacon it receives from a current scene member.

4. Realizing Scenes on Resource-Constrained Sensors

Client devices can use the Java interface from Figure 3 to interface with embedded devices. Software on these embedded devices must also support the scene abstraction. In this section, we describe this implementation, showing how the code is structured to support dynamic, opportunistic communication.

4.1. A Structured Implementation Strategy

Our implementation uses the *strategy pattern* [7], in which algorithms can be chosen at runtime depending on system conditions. The strategy pattern provides a means to define a family of algorithms, encapsulate each one, and make them interchangeable. In the scene, the clients that employ the strategies are the queries, and the different strategies are `SceneStrategy` algorithms. Figure 5 shows the resulting architecture. We decouple the scene construction from the code that implements it so we can vary message dissemination without modifying application-level query processing (and vice versa).

The remainder of this section describes one implementation of the `SceneStrategy`, the `BasicScene`, which pro-

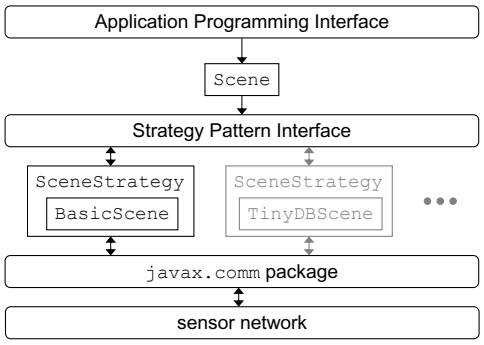


Figure 5. Simplified software architecture diagram

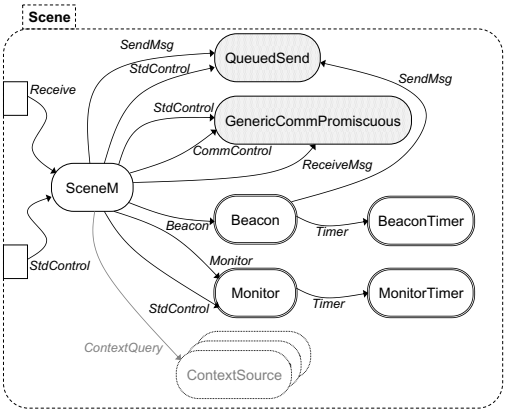


Figure 6. Implementation of the `scene` functionality on sensors

vides a prototype of the protocol's functionality. Other communication styles can be swapped in for the `BasicScene` (for example one built around TinyDB [22] or directed diffusion [12]). By defining the `SceneStrategy` interface, we enable developers who are experts in existing communication approaches to create simple plug-ins that use different query communication protocols and yet still take advantage of the scene abstraction and its simplified programming interface.

4.2. A Basic Instantiation

While the scene abstraction is independent of the particular hardware used to support it, in our initial implementation, these software components have been developed for Crossbow Mica2 motes [5] and are written for TinyOS [10] in the nesC language [8]. In nesC, an application consists of *modules* wired together via shared interfaces to form *configurations*. Figure 6 depicts the components of the scene configuration and the interfaces they share.

This implementation functions as a routing component on each node, receiving each incoming message and processing it as our protocol dictates. The `Scene` configuration uses the ReceiveMsg interface (provided in TinyOS) which allows the component to receive incoming messages. The structure of the messages received through this process is shown in Figure 7.

While the model allows a scene to be defined by multiple constraints, a single `SceneMsg` contains information about only one constraint. This is a limitation of our proof-of-concept implementation that will be removed in a more mature implementation. The sequence number is actually a combination of the unique client device and the device's sequence number. The implementation uses a static list of "types" for the metrics and the path cost functions that can be used. A message contains two constants: the metric (e.g., `SCENE_DISTANCE` or `SCENE_LATENCY`) and the path cost function (e.g., `SCENE_DFORMULA` or `SCENE_MAX`). The use of constants to specify the metric and cost function makes the implementation a little inflexible because the set of metrics must be known *a priori*, but the approach prevents messages from having to carry code. Future work will enable this automatic code deployment. The `metricValue` in the `SceneMsg` carries the previous node's calculated value for the specified metric and is updated at the receiving node. The `previousHop` in the `SceneMsg` allows this node to know its parent in the routing tree and enables scene maintenance. The `maintain` flag indicates if the query is long-lived (and therefore whether or not the scene should be maintained). Finally, `data` carries the application message.

When the `SceneM` module receives a message it has not received before, it first determines whether or not the local node should be considered as part of the scene. To do so, `SceneM` calculates the metric value for this node based on the `metricValue` in the received message and the metric and path cost function. Depending on the metric type, this may require the use of *ContextSources*, which provide relevant data (particular to each metric) for calculating a node's value. For example, a hop-count based scene requires no context source; `SCENE_HOP_COUNT` indicates that the local metric value is "1" and the path cost function `SCENE_SUM` indicates that this value should be added to the `metricValue` carried in the message. On the other hand, `SCENE_DISTANCE` indicates the local metric value is the node's location, which is implemented as a *ContextSource* that stores the node's location. When the local value for the metric has been retrieved, the `costFunction` from the message is called, for example `SCENE_DFORMULA` calculates the distance between this node and the originating node

```
typedef struct SceneMsg{
  uint16_t seqNo
    //message sequence number
  uint8_t metric
    //constant selector of metric
  uint8_t costFunction
    //constant selector of cost function
  uint16_t metricValue
    //current calculated value of metric
  uint16_t threshold
    //cutoff for metric calculation
  uint16_t previousHop
    //the parent of this node
  uint8_t maintain
    //whether the query is persistent
  uint8_t data [(TOSH_DATA_LENGTH-11)]
    //the query
}
```

Figure 7. `SceneMsg` **definition**

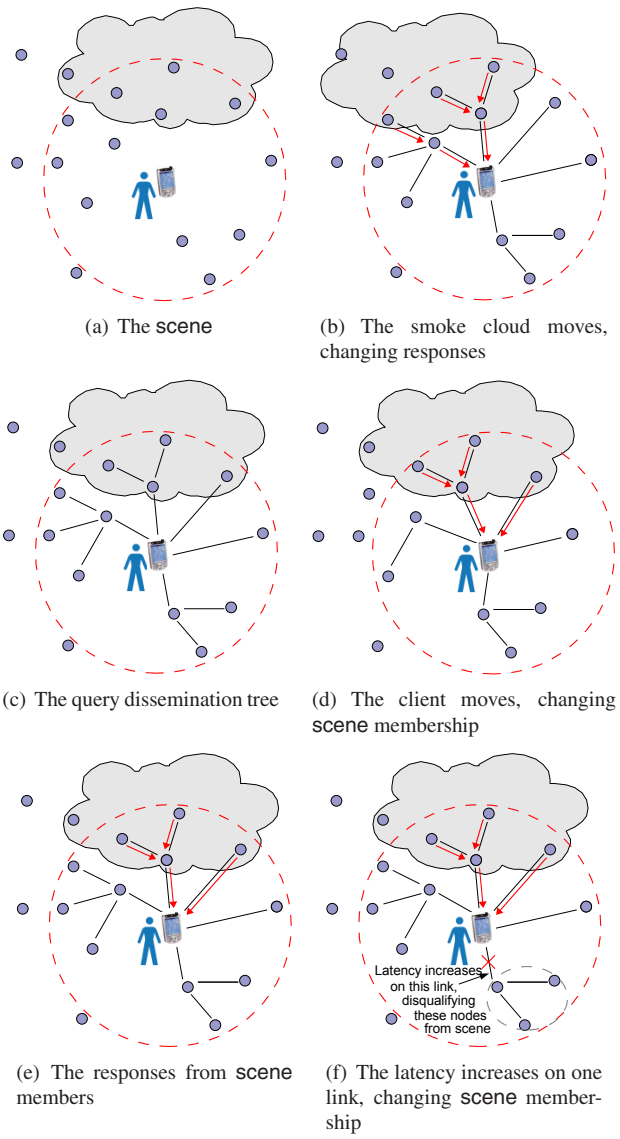
(whose location is carried in the `metricValue`). If the scene metric demands a context source that the local node does not provide (e.g., the local device has no location sensor), then the device is not considered a part of the scene.

If this node is within the scene, the message is forwarded to allow inclusion of additional nodes. To do this, a new message is created in which the node replaces the `previousHop` field with its node id. The `metricValue` field is populated according to the type of the metric; in the case of `SCENE_HOP_COUNT`, the `metricValue` is the total number of hops traversed so far (as calculated by this node adding one to the previous `metricValue`), while in the case of `SCENE_DISTANCE`, the `metricValue` is always the location of the originating node. This new message is broadcast to all neighbors, and the node also passes `data` to the application so it can answer the client device's request.

While the above is sufficient to construct a scene for obtaining a response to a one-time query, the scene needs to be maintained in the case of a persistent query. If the `maintain` flag is set, the node's `Beacon` module is activated to transmit periodically to other nodes. In addition, `SceneM` must monitor incoming beacon messages from the parent. Such messages are passed to the `Monitor`, which uses incoming beacon messages from the parent, information about the scene (from the initial message), and information from the context sources to monitor whether this node remains in the scene. If the parent has not been heard from in the last three beacon intervals, the received beacon pushes the node out of the scene, or information from the local context sources changes the node's membership, the `Monitor` generates an event that ultimately ceases the node's participation in the scene.

5. An Example Scene

In this section, we tie the code that the application developer writes in Java to what happens in the middleware on the client device and, through the scene communication protocol, on the sensor nodes. We follow a query from the application developer's hands all the way into the network and back. We assume a first responder would like to periodically receive any reading within 5m that can be delivered in less than 15ms in which the quantity of combustion products in the air exceeds 3% obscuration per meter.



(a) The scene

(b) The smoke cloud moves, changing responses

(c) The query dissemination tree

(d) The client moves, changing scene membership

(e) The responses from scene members

(f) The latency increases on one link, changing scene membership

Figure 8. Scene dynamics

Step 1: Declare a Scene. This first step uses the interface described in Section 3. Nothing happens involving network communication until the application actually uses the scene. For our application example, the application developer uses the code shown in Figure 4 to define a scene that includes every sensor (not just those measuring smoke conditions) within 5m of the declaring device and with response latency less than 15ms. (Figure 8(a) shows the nodes that will fall in the scene, if a message is distributed to them.)

Step 2: Create a query. In our example, the `AppMessage` is defined by two `Constraints`. For simplicity, we assume the application-level query processing uses constraints similar to scene definitions. In actuality, the scene communication protocol can deliver application messages of any form to all scene members, including, for example, middleware messages in a sensor network middleware [15]. The first of the constraints in the `AppMessage` requires the sensor used to support a smoke detector. The second constraint limits the sensors that respond to only those that measure a smoke condition of more than 3% obscuration per meter. *Every* sensor in the scene that has a smoke sensor periodically evaluates the query, but only responds if and when the smoke condition sensed exceeds 3% obscuration per meter. After creating this `AppMessage`, the application developer dispatches it using the previously created scene.

Step 3: Construct and Distribute Protocol Query. The middleware transforms the application's request into a protocol data unit for the scene. The resulting message carries the information about scene membership constraints *and* the data query. By its definition, the communication protocol ensures that the data query is delivered to only those sensor nodes that satisfy the scene's constraints. Thus, exactly the sensors within 5m and with a latency less than 15ms will receive the query. The query propagation stops once a node is reached whose distance from the user exceeds 5m or a node is reached whose latency exceeds 15ms (Figure 8(b) shows the dissemination tree; nodes within the circle now know they are scene members).

Step 4: Scene Query Processed by Remote Sensor. When the communication protocol receives and processes a scene message, nodes within the scene pass the received message to the application. In this example, the application layer sends periodic responses to the client using basic multihop routing if the value exceeds 3% obscuration per meter. (In Figure 8(c), the red arrows indicate return paths.) If the smoke condition is not originally greater than the threshold, the node only starts responding if the 3% obscuration per meter level is reached (as shown in Figure 8(d), where the set of nodes responding changes when the smoke cloud moves). When a node is no longer in a scene, the scene communication implementation creates a null message that it sends to the application layer to ensure that it ceases communication with the client device. Since the first responder demands periodic results, the scene has to be maintained until the node ceases to hear from its parent, the node's distance from the user exceeds 5m due to client mobility (Figure 8(e)), or the latency to a node on the path exceeds 15ms (Figure 8(f)).

As this example demonstrates, the scene abstraction is able to seamlessly support client mobility as well as changing dynamics in the network.

6. Evaluation and Analysis

In this section, we provide an evaluation of our implementation. Our protocol's intent and behavior differ significantly from other approaches, so direct comparison to existing protocols is not very meaningful. Instead, we measured our protocol's overhead in varying scenarios, by employing TOSSIM [17], a simulator that allows direct simulation of code written for TinyOS.

6.1. Simulation Settings

In generating the following results, we used networks of 100 nodes, distributed in a 200 x 200 foot area, with a single client device moving among them. We used two types of topologies: 1) a regular grid pattern with 20 foot internode spacing and 2) a uniform random placement. While the sensor nodes remained stationary, the client moved among them according to the random waypoint mobility model [13] with a fixed pause time of 0. To model radio connectivity of the nodes, we used TOSSIM's empirical radio model [29], a probabilistic model based on measurements taken from real Mica motes. In all cases, as the client moves, the scene it defines updates accordingly. In the different simulations, the client either remains stationary or moves at 2mph, 4mph, or 8mph (e.g., 4mph is a brisk walk). In these examples, scenes are defined based on the number of hops relative to the client device, ranging from one to three hops. Other metrics can be easily exchanged for hop count; we selected it as an initial test due to its simplicity. A final important parameter in these measurements is the beacon interval. We have currently set the beacon interval to be inversely proportional to client speed. This is not how beacon intervals will actually be assigned and future work will investigate better ways of assigning this value. Future work will determine the optimal beacon interval, and this information can be included in the scene building packets, allowing nodes to adapt the beacon interval depending on the application's situation.

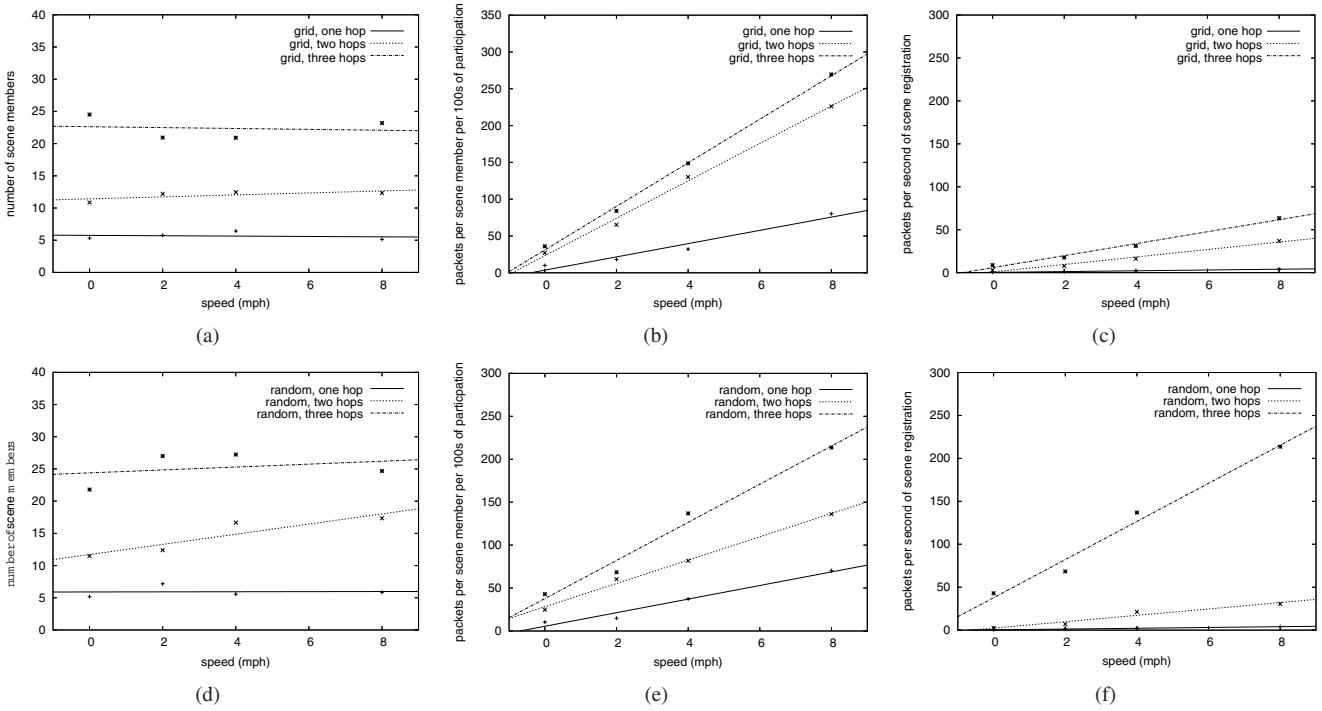


Figure 9. Simulation results

6.2. Performance Metrics

We have chosen three performance metrics to evaluate our implementation: (i) the average number of scene members, (ii) the number of messages sent per scene member, and (iii) the number of messages sent per unit time. We evaluate these metrics for both grid and random topologies.

The first metric measures how well our selected beacon intervals perform. The latter two metrics measure the scalability of the scene abstraction, i.e., how the protocol will function in scenes of increasing sizes and client mobility. The number of messages sent per scene member measures a sensor node's *cost of participation*, which also estimates the potential battery dissipation for the sensors that participate in the scene (since energy expended is proportional to radio activity). The number of messages sent per unit time is a measure of the network's *average activity*. Since the scene protocol operates on-demand, activity takes place only within the scene.

6.3. Simulation Results

Figures 9(a) and (b) show the average number of scene members as a function of client device mobility and scene size for grid and random topologies, respectively. The number of scene members is almost independent of the client node's speed. This means that the device is able to accurately reach the nodes that need to be members of its scene and shows that setting the beacon frequency to be proportional to the client node's speed accurately keeps track of the moving client.

Figures 9(c) and (d) show the number of messages sent per scene member as a function of client mobility and scene size. Because we have set the beacon frequency to be directly proportional to the speed of the client node (e.g., if the client speed is 4 mph, beacons are sent every 0.5s, if the client speed is 8 mph, beacons are sent every 0.25s), beacons are sent more frequently as speed increases, yielding the linear relationship.

Figures 9(e) and (f) show the number of messages sent per unit time as a function of mobility and scene size. Beacons are sent more frequently as the client node speed increases, causing more messages to be packed into a given time interval. In addition, as the scene size increases, more nodes become scene members, increasing the number of nodes that subsequently send beacon messages per unit time.

These results demonstrate, however, that even as the scene size increases the overhead of creating a local communication neighborhood is manageable and localized to a particular region of interest.

7. Related Work

Previous work has investigated group and neighborhood abstractions in both mobile ad hoc networks and sensor networks. In mobile ad hoc networks, the network abstractions model [27] allows applications to provide metrics over network paths. Nodes to which there exists a path satisfying the metric are included in the specifier's *network context*, while those outside are excluded. This approach is overly expressive, making it difficult to specify simple metrics. In addition, the protocol does not function on resource-constrained nodes. SpatialViews [24] abstracts properties of mobile ad hoc networks to enable applications to be developed in terms of virtual networks defined by characteristics of the underlying physical network. SpatialViews focuses on the distributed computations that occur in the defined virtual networks, while the scene abstraction focuses instead on the underpinnings of communication. Collaboration groups, defined as part of *state-centric programming* [18], formally define groups based on shared data and physical properties. The focus of collaboration groups is defining a programming model, while our approach defines the communication model necessary to efficiently support such abstractions.

In sensor networks, several approaches provide neighborhood or regional abstractions to scope applications' interactions. Hood [30] allows sensor nodes to define neighborhoods of coordination around themselves based on network properties. The implementation only allows neighborhoods that extend a single hop, while scenes allow multiple-hop neighborhoods. In addition, the scene abstraction allows dynamic updates to its participants. Abstract Regions [29] define regions of coordination and couple the abstraction with programming constructs that allow applications to issue operations over the regions. Likewise, *logical neighborhoods* [23] provide a communication infrastructure that groups logically similar nodes. These approaches do not directly consider the dynamics of mobility, and they require proactive behavior by all sensors all the time. The scene abstraction, on the other hand, uses proactivity *only* when maintaining scenes and *only* by nodes that know they are in the scene, thus reducing overhead.

While the above approaches do not address dynamics, a few constructs have begun to do so. Mobicast [11] defines a message dissemination algorithm for pushing messages to nodes that fall in a region in front of a moving target. MobiQuery [20] also supports spatiotemporal queries and allows a query area to respond to a user's announced motion profile. These approaches require nodes to have a fine-grained knowledge of their physical locations. This is not a reasonable assumption for future pervasive computing networks in which the sensor nodes and their deployments must be inexpensive and require minimal setup and administration. The scene abstraction does not require motion profiles and functions based on only logical notions of relative location. EnviroTrack [1] is tailored to providing object tracking by a dynamic group of sensor nodes. This system focuses on identifying and labeling tracked objects so that they can be addressed using more traditional communication. EnviroTrack has been extended by EnviroSuite [21] and its associated communication protocol [3]. Communication again relies on each node knowing its exact physical location. This is an acceptable assumption in these systems, given that the goal is often to know the physical location of some tracked object. In pervasive computing, however, relative locations often suffice to support applications' group formation.

Our scene abstraction learns from existing work but focuses specifically on creating a communication paradigm that supports pervasive computing applications. This requires a protocol that provides a facility for enabling direct, opportunistic interactions among heterogeneous devices and sensors. In addition, our protocol is designed with the specific intent of supporting the mobility of this regional abstraction without relying on knowledge of absolute locations.

8. Conclusion

In this paper, we have described scenes, a communication protocol that enables opportunistic communication for pervasive applications. The scene provides an efficient and dynamic approach for limiting the scope of an application's interactions to include only the embedded nodes that can provide the data the application needs, given the information-rich environment. We have presented the abstraction, its implementation, and an initial performance evaluation that demonstrates the protocol's scalability in the face of client mobility and and scene size. Future work will include a complete network performance evaluation that measures query response times and direct energy usage and a real-world deployment on a mixture of embedded and client devices.

Acknowledgments

The authors thank the anonymous reviewers for their comments. We would also like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded, in part, by the National Science Foundation (NSF), Grant #CNS-0620245 and Air Force, Grant #FA8750-06-1-0091. The conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *Proc. of ICDCS*, pages 582–589, 2004.

[2] S. Bae, S.-J. Lee, W. Su, and M. Gerla. The design, implementation, and performance evaluation of the on-demand multicast routing protocol for multihop wireless networks. *IEEE Network*, 14(1):70–77, January/February 2000.

[3] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks. In *Proc. of MobiSys*, pages 201–214, 2003.

[4] C.-C. Chiang, M. Gerla, and L. Zhang. Adaptive shared tree multicast in mobile wireless networks. In *Proc. of GLOBECOM*, pages 1817–1822, 1998.

[5] Crossbow Technologies, Inc. `http://www.xbow.com`, 2006.

[6] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, January 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In *Proc. of PLDI*, pages 1–11, 2003.

[9] M. Hewish. Reformatting fighter tactics. *Jane's International Defence Review*, pages 46–52, June 2001.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, pages 93–104, 2000.

[11] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal multicast in sensor networks. In *Proc. of SenSys*, pages 205–217, 2003.

[12] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heideman, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, February 2003.

[13] D. Johnson, D. Maltz, and J. Broch. DSR: the dynamic source routing protocol for multi-hop wireless ad hoc networks. In C. Perkins, editor, *Ad Hoc Networking*, chapter 5, pages 139–172. Addison-Wesley, 2001.

[14] C. Julien, J. Hammer, and W. O'Brien. A dynamic architecture for lightweight decision support in mobile sensor networks. In *Proc. of the Wkshp. on Building Software for Pervasive Comp.*, 2005.

[15] S. Kabadayi, C. Julien, and A. Pridgen. DAIS: enabling declarative applications in immersive sensor networks. In *Technical Report TR-UTEDGE-2006-000*, 2006.

[16] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter. The aware home: a living laboratory for ubiquitous computing research. In *Proc. of the 2nd Int'l. Workshop on Cooperative Buildings, Integrating Information, Organization and Architecture*, pages 191–198, 1999.

[17] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. of SenSys*, pages 126–137, 2003.

[18] J. Liu, M. Chu, J. Reich, J. Liu, and F. Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, October-December 2003.

[19] K. Lorincz, D. Malan, T. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, October 2004.

[20] C. Lu, G. Xing, O. Chipara, C.-L. Fok, and S. Bhattacharya. A spatiotemporal query service for mobile users in sensor networks. In *Proc. of ICDCS*, pages 381–390, 2005.

[21] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. EnviroSuite: an environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems*, 5(3):543–576, August 2006.

[22] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. on Database Systems*, 30(1):122–173, March 2005.

[23] L. Mottola and G. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proc. of InterSense*, 2006.

[24] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. of PLDI*, pages 249–260, 2005.

[25] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers. In *Proc. of SIGCOMM*, pages 234–244, 1994.

[26] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. In *Proc. of WMCSA*, pages 90–100, 1999.

[27] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proc. of ICSE*, pages 363–373, 2002.

[28] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–101, September 1991.

[29] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. of NSDI*, pages 29–42, 2004.

[30] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. of MobiSys*, pages 99–110, 2004.