Scenes: Abstracting Interaction in Immersive Sensor Networks

Sanem Kabadayi
Christine Julien

TR-UTEDGE-2007-007

Scenes: Abstracting Interaction in Immersive Sensor Networks

Sanem Kabadayı * and Christine Julien

Department of Electrical and Computer Engineering, 1 University Station C5000, The University of Texas at Austin, Austin, Texas 78712-0240, USA

Abstract

Pervasive computing deployments are increasingly using sensor networks to build instrumented environments that provide local data to immersed mobile applications immersed. These applications demand opportunistic and unpredictable interactions with local devices. While this direct communication has the potential to reduce both overhead and latency, it deviates significantly from existing uses of sensor networks that funnel information to a static central collection point. This pervasive computing driven perspective demands new communication abstractions that enable the required direct communication among mobile applications and embedded sensors. This paper presents the scene abstraction, which allows immersed applications to create dynamic distributed data structures over the immersive sensor network. A scene is created based on application requirements, properties of the underlying network, and properties of the physical environment. This paper details our work on defining scenes, providing an abstract model, an implementation, and an evaluation.

Key words: Sensor networks, coordination, pervasive computing

1 Introduction

Sensor networks, consisting of battery-powered devices that communicate wirelessly to collect information, have emerged as an integral component of pervasive environments. Much of the existing work focuses on application-specific networks where the nodes are deployed for a particular task, and data is collected at a central location to be processed and/or accessed via the Internet.

* Corresponding author. Tel.: +1 512 232-5671; fax: +1 512 471-5120.
E-mail addresses: {s.kabadayi, c.julien}@mail.utexas.edu

We envision a scenario in which sensors are general-purpose and reusable in support of pervasive computing. While the networks may remain domain-specific, the deployed applications may not be known *a priori* and may include varying adaptive behaviors, for example in aware homes [1], intelligent construction sites [2], and first responder deployments [3]. These applications require on-demand access to local information, which is exactly the vision of pervasive computing [4], in which sensor networks are integral [5].

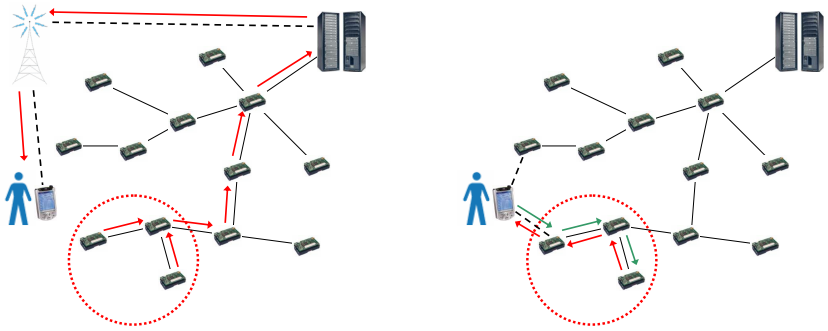
In current pervasive applications, communication is accomplished using mobile ad hoc routing protocols [6–8]. Because these protocols are tailored for providing routes across networks that can grow very large, they do not favor the local interactions common in pervasive computing. In addition, the protocols 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. Instead, applications rely on context-aware interactions, and the specific devices with which an application interacts are likely to change as the application's situation changes.

This paper introduces the scene abstraction and a protocol that provides the abstraction for developers. The scene abstraction allows an application's operating environment to include a dynamic set of embedded devices. As the device on which the application is running moves, the scene automatically updates to reflect changing conditions, thereby enabling consistent access to locally available data. We focus on supporting applications in which *client devices* (e.g., laptops or PDAs) interact directly with networks of embedded devices. While this 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 include fixed sensors that are present in environments regardless of crises and ad hoc deployments of sensors that responders may distribute when they arrive. Mobile devices include those moving within vehicles, carried by responders, and even autonomous robots for exploration and reconnaissance.

In this paper, Sections 2 and 3 motivate the problem and overview existing approaches. Section 4 characterizes our abstraction, and Section 5 describes our implementation. Section 6 provides an example definition and use of a scene, and Section 7 evaluates the performance of our approach. Sections 8 and 9 provide discussions and conclusions.

2 Problem Definition and Motivation

In immersive sensor networks, applications need to interact directly with devices embedded in the environment. This allows applications to operate over



(a) (b)

Fig. 1. Comparison of (a) existing environments with (b) the scene abstraction

information collected directly from the local area (as shown in Fig. 1(b)). This is in contrast to existing deployments in which the networks are commonly accessed through a single collection point (as shown in Fig. 1(a)). The protocols available to support communication and coordination in these networks are tailored to application situations like those depicted in Fig. 1(a). While these existing 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 new style of interaction is a direct motivation for a reexploration of protocol and coordination issues in immersive pervasive computing environments and introduces several unique challenges and heightens existing ones:

- *Locality of interactions*: An application on a client device interacts directly with local embedded devices, which can minimize communication overhead and latency. However, such a direct interaction approach can also be cumbersome with respect to enabling the 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 device carried by a mobile user. The device's connections to particular sensors and the area from which it draws information are subject to constant change.
- *Unpredictability of coordination*: Pervasive computing demands that networks be general-purpose. As such, few *a priori* assumptions can be made about applications' needs or intentions, requiring networks to adapt to unexpected and changing situations.
- *Complexity of programming*: The desire for end-user applications (as opposed to database-oriented data collection) increases the demand for applications and the number of programmers that will need to construct them.

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

3 Related Work

The previous section detailed a set of requirements for communication constructs necessary to support general-purpose pervasive computing applications that rely on immersive sensor networks. In this section, we highlight some existing work for both sensor networks and pervasive computing and examine how well each approach addresses the aforementioned requirements.

Network abstractions [9] allows applications to provide metrics over network paths; nodes to which there exists a path satisfying the metric are included in the *network context*. This approach is overly expressive for pervasive applications, making it difficult to specify simple metrics. In addition, the protocol does not function on resource-constrained nodes. SpatialViews [10] abstracts properties of mobile ad hoc networks to enable application development in terms of virtual networks based on the underlying physical network. SpatialViews focuses on distributed computations in the virtual networks, while we focus first on defining such virtual networks and the underpinnings of communication that hold them together. Collaboration groups, defined as part of *state-centric programming* [11], abstract common patterns in application-specific communication. However, the focus is in defining a programming model, not in the communication model necessary to efficiently support it.

Several approaches also define neighborhoods in sensor networks. Hood [12] allows sensor nodes to define neighborhoods around themselves based on network properties. The implementation only allows neighborhoods that extend a single hop, while multiple-hop neighborhoods are necessary for expressive pervasive computing. Abstract Regions [13] define regions of coordination and couple the abstraction with programming constructs that allow applications to issue operations over the regions. Likewise, *logical neighborhoods* [14] provide a communication infrastructure that logically groups similar nodes. These three approaches do not directly consider the dynamics of mobility, and they require proactive behavior by all sensors all the time.

A few constructs have also begun to address mobility. Mobicast [15] pushes messages to nodes that fall in a dynamic region in front of a moving target. MobiQuery [16] 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, which is not reasonable in future pervasive computing networks where the sensor nodes and their deployments must be inexpensive and require minimal setup and administration. In EnviroTrack [17–19], a dynamic group of sensors identify and label tracked objects so that they can be addressed using more traditional communication. 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, supporting local interactions requires specifying only the relation the nodes must have to the user and is not concerned with exact locations. For example, a particular first responder may be interested only in interacting with other responders who are nearby. Obtaining GPS coordinates and resolving locations in a global shared coordinate space would unnecessarily increase the computational tasks and the cost of a pervasive computing application.

Creating a communication paradigm that supports pervasive computing applications requires an abstraction and implementing protocol that provide a facility for enabling direct, opportunistic interactions among heterogeneous devices and sensors. This protocol needs to support the mobility of this regional abstraction without relying on knowledge of absolute locations.

4 Scenes: Declarative Local Interactions

The set of data sources near a user changes based on the user's mobility. If the network is well-connected, the user will be able to reach vast amounts of raw information. The application must limit the scope of its interactions to 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 a scene that constrains which sensors influence the application. This abstraction defines local, multihop neighborhoods surrounding a particular application, supports mobility by dynamically updating the scene's participants, and minimizes how much the developer must know about the underlying implementation. The constraints that define a scene may be on properties of hosts (e.g., battery life), network links (e.g., bandwidth), and data (e.g., type).

4.1 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 (i.e., a property of hosts, links, or data).
- *Path cost function*: A function (such as sum, average, minimum, or maximum) that operates on a network path to calculate the cost of the 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, ... , 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 a path cost function, and T a threshold.

Fig. 2 demonstrates the relationships between these components. This figure shows only a one-constraint scene and a single network path. The cost to a node in the path (e.g., node i) is calculated by applying the path cost function to the metric using 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 the metric, path cost function, and threshold, which enable each node to independently determine its scene membership. The selected network paths correspond to branches of a routing tree created as part of scene construction. When a node needs to relay a reply back to the user, the reverse of the path on the routing tree can be used. 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 metric is shorter, this path is chose and the node forwards the information again because it may enable new nodes to be included in the scene.

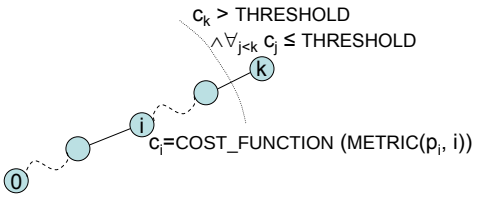


Fig. 2. Distributed scene computation

This scene definition can be formalized in the following way:

> Given a client α, a metric M, and a positive threshold T, find the set of hosts S_α such that all hosts in S_α are reachable from α and, for all hosts β in S_α, the cost of applying M on some path from α to β is less than T:
>
> $S_\alpha = \langle set\ \beta : M(\alpha, \beta) < T :: \beta \rangle$ [1]

The above formalization is a simplification of the scene abstraction; it is for a scene that uses only one metric. If the scene is specified by multiple metrics, the center expression must be true for all metric/threshold pairs.

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

The scene concept conveys a notion of locality, and each application decides how "local" its interactions need to be. A first responder leader may want to have an aggregate view of the smoke conditions over the entire site, while a particular responder may require only a scene that contains readings only from nearby sensors. 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 in the scene may change.

4.2 A Programming Interface for Scenes

To present the scene 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 specify scenes through a Java programming interface, and these specifications are translated into low-level sensor code. Fig. 3 shows the Java API, which relies on a `Query` that the application provides when interacting with a `Scene`. This `Query` should be delivered to every member of the `Scene`. The `ResultListener` interface used in the `Scene` API allows nodes who are members of the scene to return responses to the client device. The `Scene` API intentionally does not restrict what kind of application-level communication these query and reply interactions can encode; this depends on the particular application layer running on both the client device and the sensor. The scene abstraction simply provides expressive connectivity among these components. In Section 6, we will explore a simple application layer for a first responder application example.

```
class Scene{
   public Scene(Constraints[] c);
   public void send(Query q, ResultListener rl);
   public void maintain(Query q, ResultListener rl, int frequency);
}
```

Fig. 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 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. The `frequency` parameter in the `maintain` method indicates how often the application expects responses.

The `Scene` API is intentionally simple. It focuses on providing access to the scene constructs and not on incorporating client functionality into the scene communication components. We are motivated to keep the API as slim as possible to ease the implementation on resource-constrained devices. By limiting the functionality available to applications, we more closely match the capabilities of this underlying constrained hardware. While only the one-time query behavior is essential, the `maintain` operation enables a more efficient implementation of persistent queries. This is especially important in resource-constrained networks, where minimizing communication overhead is essential.

Fig. 4 shows examples scene metrics. The simplest metric (not shown), SCENE_HOP_COUNT, assigns a value of one to each network link. 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. It is possible to extend the number of constraints inductively (for example, the hop count scene above could be further restricted using latency as a second constraint), and this combinatorial nature provides significant flexibility to the programmer.

	Battery Power Scene
Metric	SCENE_BATTERY_POWER
Aggregator	SCENE_MIN
Metric Value	min battery power
Threshold	min permissible battery power
	Distance Scene
Metric	SCENE_DISTANCE
Aggregator	SCENE_DISTANCE
Metric Value	location of source
Threshold	max physical distance
	Latency Scene
Metric	SCENE_LATENCY
Aggregator	SCENE_SUM
Metric Value	total latency on path so far
Threshold	max permissible latency

Fig. 4. Example Scene Definitions

4.3 Maintaining Scenes

While a scene provides the appearance of a dynamic data structure, the implementation behaves on demand; no proactive behavior occurs. Only when the application uses a scene does the protocol communicate with other local devices, reducing the overall communication overhead. At first glance, this approach may appear to incur an unpredictable latency for the first query posed to a scene. However, queries traverse the same path as the scene construction

messages, and the queries themselves carry the scene construction information. Therefore, the on-demand construction incurs no additional latency.

For one-time queries, a scene is created, and the scene information is not stored or updated in any way. On the other hand, if the scene is to be used for a persistent query, it needs to be maintained. To maintain the scene for such continuous queries, 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.

Pervasive applications expect access to locally available resources. Consider an application in an aware home. An application may connect to resources that it can monitor and control within the room the user occupies. As the user moves around the home, the scope of this control should change to match the user's changing rooms. Therefore, we provide the automatic maintenance described above instead of calculating a static scene when the application initially declares it. This style of maintenance is particularly well-suited to pervasive computing applications which demand automated context-awareness.

4.4 Defining Scenes Based on Physical Characteristics

The metrics used to specify scenes can be divided into two categories: those that define scenes based on properties of *network paths* or the devices on the network paths (e.g., latency or battery power) and those that define scenes based on *physical characteristics* of the environment (e.g., location or temperature). Using a physical characteristic to calculate network paths is plagued by the *C-shaped network problem*. Consider the network shown in Fig. 5. Nodes A and B are within 50m of each other, yet a discovery from A to B must leave the region of radius 50m surrounding A to find B. The only way to guarantee that every device is discovered is to flood the entire network.

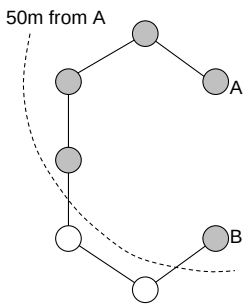


Fig. 5. A C-network

In pervasive computing networks, an application may not be in direct communication with the devices with which it needs to interact. In a first responder situation, safety applications may dictate that each user has information about

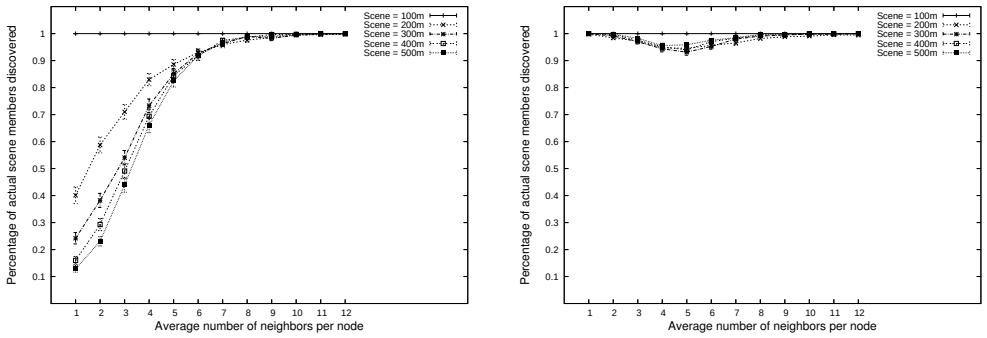


Fig. 6. Accuracy of location-based scene calculations

a region larger than a device's communication radius, for example to monitor the presence and movement of fire or gases. For this reason, we focus on building the best multi-hop neighborhoods possible. For metrics that measure physical characteristics, the question remains as to how to handle the ambiguity separating the natural specification (e.g., "all devices within 50m") and the ability of a protocol to efficiently satisfy that specification (i.e., without flooding the network). We favor an approach with a simple programming interface that allows scenes to be specified that may not be able to be exactly built (due to the presence of configurations such as that shown in Fig. 5). In such situations, our scene building protocol may not find some scene members even though they are transitively connected.

Figs. 6(a) and (b) show the results of experiments that demonstrate the ramifications of this design decision. In these experiments, we generated random network topologies in a $1000m^2$ space with the following parameters. the number of nodes was randomly selected to be between 20 and 400, and each node was randomly placed. We used a communication radius of 100m, i.e., any two nodes within 100m of each other were considered "neighbors." We constructed scenes based on physical distances ranging from 100m to 500m. A 100m scene includes only nodes within the requester's communication range (i.e., within one-hop). In each graph, the x-axis shows the average number of one-hop neighbors per node. Each point corresponds to 500 samples, and 95% confidence intervals are given. For each sample, one node was randomly selected to request a scene of the specified size.

Fig. 6(a) shows the percentage of *actual* scene members discovered by our protocol. This includes every node within the specified physical distance radius, even nodes to which no network connectivity exists. At low network density, the quality of the scene construction was poor, especially as the physical size of the scene increased. This is because the network was so sparsely connected that it was unlikely that nodes were able communicate, especially when they desired to find other nodes at large distances. However, with increasing density, our protocol found more than 90% of the actual scene members.

Fig. 6(b) demonstrates even stronger motivation for our best-effort approach for scenes based on physical characteristics. This graph limits the error expressed to only those scene members that were not discovered but were connected by a finite number of network hops (i.e., those nodes reachable by flooding). The percentage of scene members that our method *did not* discover is never more than 10% and is usually close to 0. The valley corresponds to cases when the network was largely connected but connections were sparse. In these situations, roundabout paths may exist when more direct routes do not. To the left of the valley, the network was largely disconnected, so we do not miss many connected scene members; to the right, the network was much more connected, and the direct approach is quite successful.

The results demonstrate that our approach tends to find the vast majority of the scene members under reasonable conditions. Therefore, we favor natural scene specifications over complete accuracy of scene membership.

5 Realizing Scenes on Resource-Constrained Sensors

The model for both construction and maintenance of scenes described in the previous section is tailored to the requirements of pervasive computing environments. In creating applications for client devices, developers can leverage the Java interface from Fig. 3, which allows them to use the scene communication abstraction 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.

5.1 A Structured Implementation Strategy

Our implementation uses the *strategy pattern* [20], a software design pattern in which algorithms (such as strategies for scene construction and maintenance) 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.

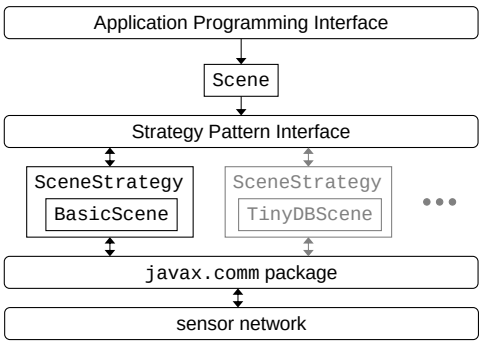


Fig. 7. Simplified software architecture

Such an approach allows the algorithms to vary independently from the clients that use them. In the scene, the clients that employ the strategies are the

queries, and the different strategies are `SceneStrategy` algorithms. Fig. 7 shows the resulting architecture. We decouple the `scene` construction from the code that implements it so that 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 provides a prototype of the protocol's functionality. Other communication styles can be swapped in for the `BasicScene` (for example one built around TinyDB [21] or directed diffusion [22]). 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.

5.2 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 [23] and are written for TinyOS [24] in the nesC language [25]. Our nesC implementation of the `scene` abstraction (along with other project information) is available at `http://mpc.ece.utexas.edu/scenes/index.html`. In nesC, an application consists of *modules* wired together via shared interfaces to form *configurations*. Fig. 8 depicts the components of the `scene` configuration and the interfaces they share.

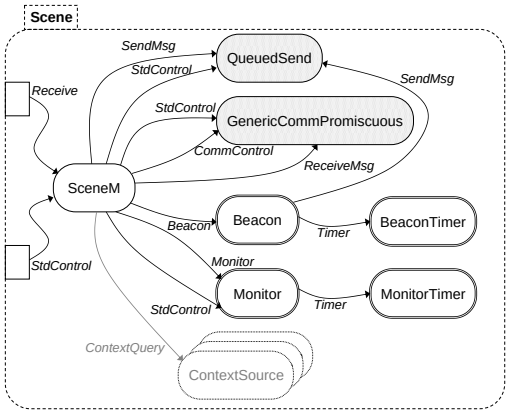


Fig. 8. Implementation of the `scene` on sensors

This implementation functions as a routing component on each node, receiving each incoming message and processing it as our protocol dictates. In this picture, we show components as rounded rectangles and interfaces as arrows connecting components. A component *provides* an interface if the corresponding arrow points towards it and *uses* an interface if the corresponding arrow points away it. If a component provides an interface, it must implement all of the *commands* specified by the interface, and if a component uses an interface, it can call any commands declared in the interface and must handle all *events* generated by the interface.

From the perspective of a message coming in from a neighboring node, The Scene configuration uses the ReceiveMsg interface (provided in TinyOS) which allows the component to receive incoming messages from the radio (by handling the receive event). Specifically, within the Scene configuration, the SceneM component handles this event. SceneM implements most of the logic of the scene implementation on the sensor. The structure of the messages received through this process is shown in Fig. 9.

```
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
}
```

Fig. 9. SceneMsg definition

While the model allows a scene to be defined by multiple constraints, a single SceneMsg contains only one constraint. This is a limitation of our proof-of-concept implementation that will be removed in a more mature implementation. The SceneMsg contains a sequence number that uniquely identifies the message. The sequence number is a combination of the unique client device id and the device's sequence number. This allows a receiving node to differentiate between scenes for different client applications. A message contains two constants that instruct the SceneM component in processing the message: 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. In the case of a scene based on location, the metricValue may be the location of the source node, while in the case of a metric based on end-to-end latency, the metricValue may be the aggregate total latency on the path the message has traveled. 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 (i.e., the Query).

Fig. 10 shows how a scene message is processed at a receiving node.

When SceneM receives a message it has not received before (based on the message's unique sequence number), it determines whether the node should be a member of the sceneby calculating the node's metric value based on the metric and path cost function. Because these fields are constants, SceneM can lookup their meanings in a table and determine how to calculate the new metric value. Depending on the metric, this may require *ContextSources*, which provide relevant data 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. If necessary, context values are retrieved from the designated ContextSource through the query command in the ContextQuery interface. If the metric demands a context source that the node does not provide (e.g., the local device has no location sensor), the device is not considered a part of the scene. When the necessary context values have been retrieved, the costFunction from the message is invoked. For example SCENE_DFORMULA calculates the distance between this node and the originating node (whose location is carried in the metricValue). The newly calculated value for the metric is compared against the value of the threshold in the SceneMsg. If the new value does not satisfy the threshold, then this node is not within the scene and the message is ignored.

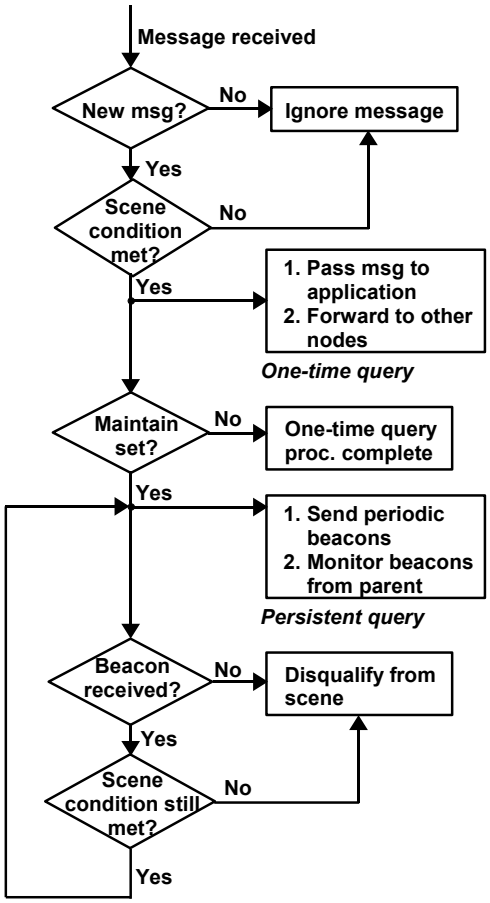


Fig. 10. Scene construction flowchart

If this node is within the scene, the message is forwarded to allow inclusion of additional nodes. 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 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 (using TOS_BROADCAST_ADDR as the destination). The node also passes data to the application (through the Receive interface shown in Fig. 8).

The scene also needs to be maintained in the case of a persistent query. If the maintain flag is set, then SceneM must monitor changes that may impact the node's membership. For example, if the scene is defined by relative location and the user is walking through the network, as he moves away from a sensor, the sensor will need to be removed from the scene. The scene implementation on the sensors uses a Beacon module to transmit periodically to other nodes. As the Monitor component (described next) detects changes in the metric value, the value is updated (through SceneM) and reflected in the beacons sent to neighbors. In addition, SceneM must monitor incoming beacon messages from the parent. Such messages are received in SceneM and passed to the Monitor. The Monitor uses beacons from the parent, information about the scene (from the initial message), and information from the context sources to monitor whether the node remains in the scene. In addition, the MonitorTimer requires that the node has heard a beacon from the parent at least once in the last three beacon intervals. If either the parent has not been heard from or the received beacon pushes the node out of the scene, the Monitor generates an event for SceneM that ultimately ceases the node's participation in the scene, including signaling the application to cancel its interactions with the client device.

6 An Example Scene

In this section, we tie the code that the application developer writes through the scene communication protocol to what happens on the sensors. We follow a query from the application developer's hands into the network and back. Within this section, we use an example application drawn from the first responder domain that assumes personnel deployed on a dangerous site that may contain smoke clouds. Specifically, 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 level of combustion products in the air exceeds 3% obscuration per meter.

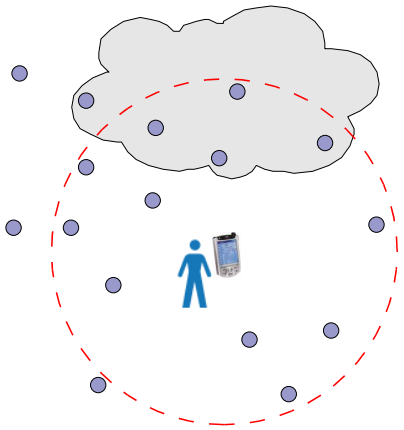


Fig. 11. The scene

```
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)} ) ;
```

Fig. 12. First responder scene construction

Step 1: Declare a Scene. This first step uses the interface described in Section 4 to declare a scene. For example, in a first responder deployment, the code in Fig. 12 defines 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 11 shows the nodes that will fall in the scene, if a message is distributed to them.

```
Query q = new Query(new Constraint(''Sensor'', Query.EQUALS_OPERATOR,
                                   ''Smoke''),
                    new Constraint(''Measurement'', Query.GT_OPERATOR,
                                   ''3'')} );
```

Fig. 13. Example first responder query construction

Step 2: Create a query. The next step is performed by the application developer using the `Query` data type in conjunction with the `Scene` instance just created. In our example, the developer creates a `Query` with two `Constraints`. For simplicity, we assume the application-level processing uses constraints similar to those used in scene definitions. In actuality, the scene protocol can deliver application messages of any form to all scene members, including, for example, middleware messages in a sensor network middleware [26]. In our example `Query`, the first of the constraints requires the sensor used to support a smoke detector. The second constraint limits the sensors that respond to the query to only those that measure a smoke condition of more than 3% obscuration per meter. The code used to construct this `Query` is shown in Fig. 13. *Every* sensor in the scene that has a smoke sensor periodically evaluates the query, but a sensor will only send a response to the client if and when the smoke condition sensed exceeds 3% obscuration per meter. After creating this `Query`, the application developer dispatches it using the previously created scene.

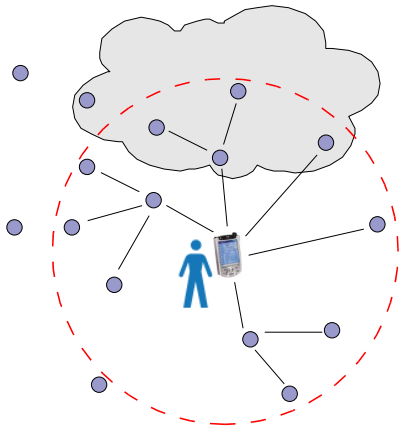


Fig. 14. The query dissemination tree

Step 3: Construct and Distribute Protocol Query. The scene implementation 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 whose latency exceeds 15ms. Fig. 14 shows the dissemination tree; nodes within the dashed circle now know they are scene members.

Step 4: Scene Query Processed by Remote Sensor. When the communication protocol running on a remote sensor receives and processes a scene message, if it determines that the node lies within the scene, it passes the received message to the application. In our example first responder scenario, our simple application layer sends periodic responses to the client if the value exceeds 3% obscuration per meter. These responses propagate using basic multihop routing.

In Fig. 15, the red arrows indicate the return paths these sensors use to return query responses to the client device. Since the first responder demands periodic results so he can monitor changes in smoke density on a site, the scene must be maintained in the face of changes. 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. Fig. 16(a) shows that this set of responding nodes may change when the smoke cloud moves. When a node is no longer in a scene, the scene communication implementation on that node creates a null message that it sends to the application layer to ensure that it ceases communication with the client device. Other changes in the network topology or physical environment can also cause scene changes. In our example, if the node's distance from the user exceeds 5m due to client mobility (Fig. 16(b)), or the latency to a node on the path exceeds 15ms (Figure 16(c)), the scene membership may have to be recalculated. As demonstrated in the figures, this may cause nodes to be removed from the scene or new nodes to be added to the scene.

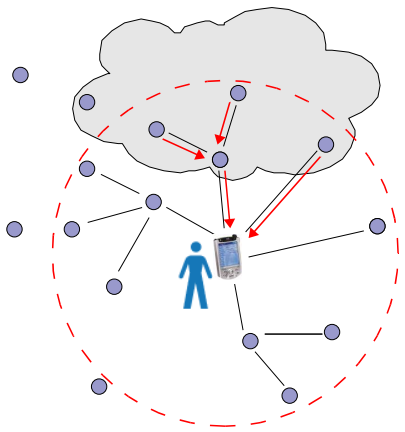


Fig. 15. The responses from scene members

Step 5: Result Received by Client Device. After propagating through the underlying communication substrate, query replies will arrive at the client device's sensor network interface. At the client device, the result is handled by the scene implementation on the sensor and passed into the Java implementation. This implementation demultiplexes the request and hands it back to

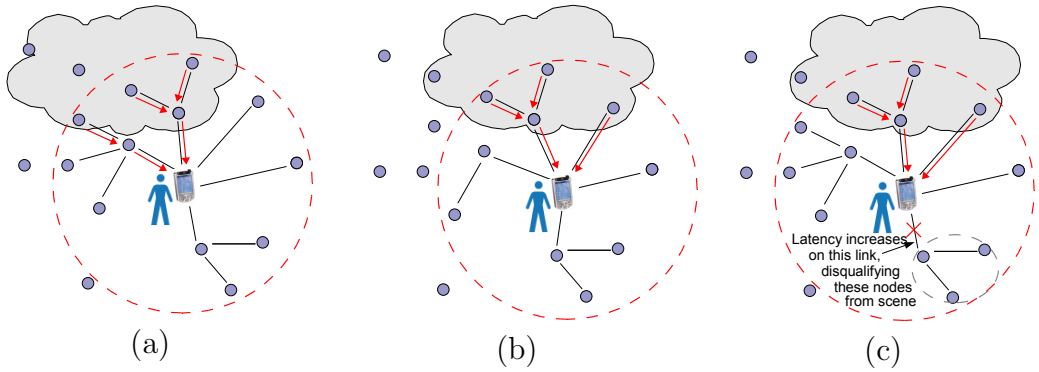


Fig. 16. Dynamics within a scene. (a) The smoke cloud moves, changing responses; (b) The client moves, changing scene membership; (c) The latency increases on one link, changing scene membership.

the appropriate application through the `ResultListener` that was provided as part of dispatching the query to the scene. At this point, control for this query reply transfers back to the client's application and its `ResultListener` which handles the query's result (or queries' results if multiple matches existed). For persistent queries, as more results arrive, the same process occurs for each received result.

As this example has demonstrated, the scene abstraction seamlessly supports client mobility within an immersive sensor network. The abstraction automatically adjusts the application's view of data in response to changes in the network or the physical environment. This context-awareness is essential to pervasive computing applications that rely on localized interactions in large-scale networks. In the next section, we provide some performance characterizations of the protocol implementing the scene abstraction to show that it provides good scalability and overhead in such resource-constrained networks.

7 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 [27], a simulator that allows direct simulation of code written for TinyOS. TOSSIM therefore allows us to perform large-scale simulations (in this case, of 100 nodes); these simulations are of a scale that is unmanageable in real sensor networks. However, the code executed in TOSSIM is the same code running on the sensors, and small scale runs in the sensors themselves corroborate the results reported here.

7.1 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 [7] with a fixed pause time of 0. To model radio connectivity of the nodes, we used TOSSIM's empirical radio model [13], 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. While these speeds appear to be on the slow side, they are reasonable for the pervasive computing scenarios we consider (e.g., 4mph is a very brisk walk; 8mph is an expected speed of vehicles on construction sites, etc.). 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. Recall that the beacon interval is the specified amount of time over which each node monitors beacons from its parent in the routing tree to maintain its own membership in the scene for continuous queries. If the node does not hear from its parent during that beacon interval, it disqualifies itself from the scene. We have currently set the beacon interval to be inversely proportional to client speed. Because this approach relies on shared global knowledge, this is not how beacon intervals will actually be assigned, and future work will investigate better ways of assigning this value. For example, in the future, clients can monitor their own speeds and embed this beacon interval in scene building packets. Alternatively, individual sensors could monitor the change in client connections over time (which can be correlated with mobility [28]) and use this information to locally adapt the beacon interval. Both options allow nodes to adapt the beacon interval depending on the particular situation; this adaptation is critical to context.

7.2 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.

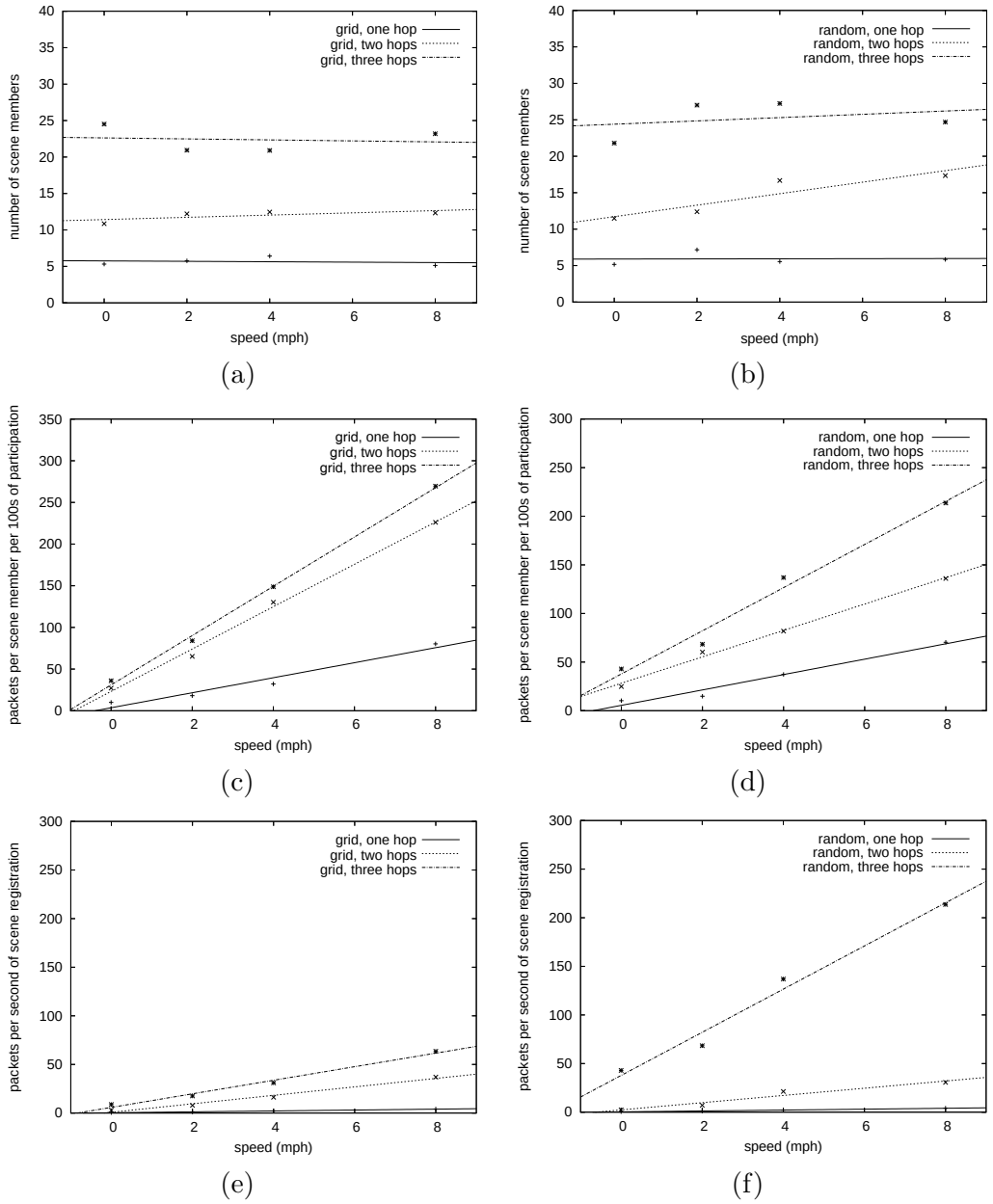


Fig. 17. Simulation results

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.

Figs. 17(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.

Figs. 17(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. This shows how the battery dissipation for each sensor that participates in the scene would scale with increasing client mobility.

Figs. 17(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 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. Since the scene protocol is an on-demand communication protocol, the activity in the network takes place only within the scene. The nodes that do not satisfy the scene constraints are inactive. The average number of scene members stays constant over changing client mobility for a specified scene size.

8 Discussions and Future Work

The evaluation reported in the previous section demonstrates the feasibility of placing a data communication abstraction for pervasive computing on highly resource-constrained devices. This is an important step as such embedded devices are an essential part of any immersive sensor network that supports such applications. The quantitative evaluation in the previous section, however, does not address the relationship of the scene abstraction to

other neighborhood protocols. This is largely due to the fact that the client-centered, device-agnostic form of the scene abstraction provides a completely novel perspective on interactions in sensor networks. Here, we briefly provide a further quantitative comparison to other reasonably similar abstractions.

Recent sensor network communication paradigms differ significantly from approaches for even mobile networks in that they include constructs to directly address resource constraints and to enable cooperation among nodes. Directed diffusion [22], for example provides a decentralized data-centric communication protocol that allows nodes in the network to cooperate to aggregate data as it is funneled back to the requester. However, in directed diffusion, this gradient setup is expensive, and the paths established should last a long time to amortize this cost over their usage. This makes it difficult for directed diffusion to cope with unpredictable resource availability and almost impossible to handle the types of mobility and dynamics that pervasive computing environments exhibit.

The regional abstractions [12–14] described in Section 3 provide a closer match to the scene abstraction's capabilities. However, these approaches still view sensor networks as supporting facilities for remote distributed sensing. As such, they make strong assumptions about the static parameters of the environment, especially of the sensor placement themselves, and they do not consider embedded pervasive computing clients roving among the sensors. However, all three of these approaches are reasonable points of comparison, and future work will explore more quantitative comparisons to their behavior.

Another important point of evaluation not addressed in this paper is that of the expressiveness of the scene abstraction. Specifically, future work will investigate the question of how rich and usable the abstraction is with respect to the requirements of a variety of applications. Our own work with intelligent construction sites [29] has demonstrated applicability to a second domain (other than the first responder domain used in this paper), but we plan to perform additional user and application studies to further validate our expressiveness claims. In addition, we have made our implementation available at `http://mpc.ece.utexas.edu/scenes/index.html` for others to download and try for their applications.

A final interesting point of discussion is that of the degree of adaptivity the scene abstraction provides. We have motivated throughout the paper that awareness of and adaptation to the surrounding environment are crucial to enabling pervasive computing applications. The scene abstraction as described in this paper already incorporates several points of adaptation, most specifically allowing the participants in a scene to change over time in response to client mobility or to changes in the network or physical environment. We have already discussed such adaptation with respect to setting the scene's bea-

con frequency to be sensitive to the client mobility or a sensor node's local perception of mobility. Future work will explore additional adaptation points that could make the abstraction even more responsive to pervasive computing applications. For instance, one could imagine a scene's threshold expanding or contracting based on the environmental values sensed or the density of available readings.

9 Conclusion

This paper presented the scene data communication abstraction, a new communication paradigm tailored to immersive sensor networks that support pervasive computing. Specifically, the scene abstraction is the first to support dynamic client devices roving among embedded sensors. The scene abstraction and the protocol that implements it utilize a high degree of context-awareness and adaptation. This allows an application's scene to consistently reflect its instantaneous operating environment. Using the scene abstraction and protocol, an application has a direct view of the information sources available in the immediate environment. In addition, our approach combines this local perspective with a communication protocol for disseminating client requests and returning replies from the scene participants. We have presented the abstraction, its implementation, and an initial feasibility study of its performance. Future work will include a more complete evaluation to include measurements of the approach's expressiveness through a larger-scale real-world deployment on a mixture of embedded and client devices.

Acknowledgements

The authors would 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), Grants # CNS-0620245 and OCI-0636299. The conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, W. Newstetter, The aware home: a living laboratory for ubiquitous computing research, in: Proc. of the 2^{nd} Int'l. Workshop on Cooperative

Buildings, Integrating Information, Organization and Architecture, 1999, pp. 191–198.

[2] C. Julien, J. Hammer, 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.

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

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

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

[6] C. Perkins, P. Bhagwat, Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers, in: Proc. of SIGCOMM, 1994, pp. 234–244.

[7] D. Johnson, D. Maltz, J. Broch, DSR: the dynamic source routing protocol for multi-hop wireless ad hoc networks, in: C. Perkins (Ed.), Ad Hoc Networking, Addison-Wesley, 2001, Ch. 5, pp. 139–172.

[8] C. Perkins, E. Royer, Ad hoc on-demand distance vector routing, in: Proc. of WMCSA, 1999, pp. 90–100.

[9] G.-C. Roman, C. Julien, Q. Huang, Network abstractions for context-aware mobile computing, in: Proc. of ICSE, 2002, pp. 363–373.

[10] Y. Ni, U. Kremer, A. Stere, L. Iftode, Programming ad-hoc networks of mobile and resource-constrained devices, in: Proc. of PLDI, 2005, pp. 249–260.

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

[12] K. Whitehouse, C. Sharp, E. Brewer, D. Culler, Hood: a neighborhood abstraction for sensor networks, in: Proc. of MobiSys, 2004, pp. 99–110.

[13] M. Welsh, G. Mainland, Programming sensor networks using abstract regions, in: Proc. of NSDI, 2004, pp. 29–42.

[14] L. Mottola, G. Picco, Programming wireless sensor networks with logical neighborhoods, in: Proc. of InterSense, 2006.

[15] Q. Huang, C. Lu, G.-C. Roman, Spatiotemporal multicast in sensor networks, in: Proc. of SenSys, 2003, pp. 205–217.

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

[17] 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, A. Wood, EnviroTrack: towards an environmental computing paradigm for distributed sensor networks, in: Proc. of ICDCS, 2004, pp. 582–589.

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

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

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

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

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

[23] Crossbow Technologies, Inc., `http://www.xbow.com` (2006).

[24] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, in: Proc. of ASPLOS, 2000, pp. 93–104.

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

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

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

[28] R. Srinivasan, C. Julien, Passive network awareness for adaptive mobile applications, in: Proc. of the 3^{rd} Int'l. Wkshp. on Managing Ubiquitous Communications and Services, 2006, pp. 22–31.

[29] J. Hammer, I. Hassan, C. Julien, S. Kabadayi, W. O'Brien, J. Trujillo, Dynamic decision support in direct-access sensor networks: A demonstration, in: Proc. of MASS, 2006.