TinyLIME: Bridging Mobile and Sensor Networks through Middleware

Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy

Amy L. Murphy
Dept. of Informatics
University of Lugano, Switzerland
amy.murphy@unisi.ch

Gian Pietro Picco
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
picco@elet.polimi.it

Abstract

In the rapidly developing field of sensor networks, bridging the gap between the applications and the hardware presents a major challenge. Although middleware is one solution, it must be specialized to the qualities of sensor networks, especially energy consumption. The work presented here provides two contributions: a new operational setting for sensor networks and a middleware for easing software development in this setting. The operational setting we target removes the usual assumption of a central collection point for sensor data. Instead the sensors are sparsely distributed in an environment, not necessarily able to communicate among themselves, and a set of clients move through space accessing the data of sensors nearby, yielding a system which naturally provides context relevant information to client applications. We further assume the clients are wirelessly networked and share locally accessed data. This scenario is relevant, for example, when relief workers access the information in their zone and share this information with other workers. Our second contribution, the middleware itself, is an extension of LIME*, our earlier work on middleware for mobile ad hoc networks. The model makes sensor data available through a tuple space interface, providing the illusion of shared memory between applications and sensors. This paper presents both the model and the implementation of our middleware incorporated with the Crossbow Mote sensor platform.*

1. Introduction

Wireless sensor networks have emerged as a novel and rapidly evolving field, with staggering enhancements in performance, miniaturization, and capabilities. However, we observe that the features provided by the computing and communication hardware still await to be matched by an appropriate software layer enabling programmers to easily and efficiently seize the new opportunities.

At the same time, we observe that much of the work in the area assumes statically deployed sensors organized in a network from which data is collected and analyzed at a central location. For many applications, e.g., habitat monitoring [16], this is a natural setting. However, in many others this seems to be overly constraining. First of all, it may be impractical or even impossible to choose where to place the collection point, for example in disaster recovery or military settings. Moreover, in the centralized scenario sensors contribute to the computation independently of their location. In other words, there is no notion of *proximity* supporting, say, reading only the average temperature sensed around a technician while he walks through a plant. Finally, the traditional centralized scenario poses technical challenges related to the problem of routing the sensed information from sensors at the fringe of the system back to the collector host, by mediating the conflicting issues of multi-hop routing and power saving.

In this paper, we provide a contribution to both issues, by proposing a new operational setting for sensor network applications, as well as a middleware supporting their development. Our reference operational setting replaces centralized data collection with a set of mobile monitors interconnected through ad hoc, wireless links, and able to receive sensed data only from the sensors they are directly connected to. This way, sensors effectively provide each mobile monitor with *context-sensitive* data.

To support this novel operational setting, we extend and adapt a model and middleware called LIME [13, 15], originally designed for mobile ad hoc networks (MANETs). Changes to the model are required to match the operational setting described so far. Extension of the middleware is needed to cope with the requirements related with power consumption and with the sheer need of installing the middleware components on devices with very limited computational resources. The result of this effort, called TinyLIME, has been implemented entirely of top of the original LIME and deployed using Crossbow motes [2] as the target platform.

The paper is organized as follows. Section 2 contains background information about LIME and the mote sensors. Section 3 illustrates the operational setting we propose and target. Section 4 presents the TinyLIME model, while Section 5 describes the architecture of the corresponding middleware developed for motes. Additional implementation details and evaluation are provided in Section 6. Section 7 places our work in the context of related efforts. Finally, Section 8 ends the paper with brief concluding remarks.

2. Background

In this section we provide the reader with the necessary background about LIME, which TinyLIME builds upon, and on the Crossbow Mote sensor platform, which serves as the target deployment technology for our implementation.

2.1. LIME

TinyLIME is a data-sharing middleware based on LIME, which in turn adapts and extends towards mobility the tuple space model made popular by Linda.

Linda and Tuple Spaces. Linda [6] is a shared memory model where the data is represented by elementary data structures called *tuples* and the memory is a multiset of tuples called a *tuple space*. Each tuple is a sequence of typed fields, such as ⟨"foo", 9, 27.5⟩ and coordination among processes occurs through the writing and reading of tuples. Conceptually all processes have a handle to the tuple space and can add tuples by performing an $\mathbf{out}(t)$ operation and remove tuples by executing $\mathbf{in}(p)$ which specifies a pattern, p for the desired data. The pattern itself is a tuple whose fields contain either *actuals* or *formals*. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of ⟨"foo", ?integer, ?float⟩ are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by $\mathbf{in}$ is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive $\mathbf{rd}(p)$ operation.

Both $\mathbf{in}$ and $\mathbf{rd}$ are blocking, i.e., if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple appears. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives $\mathbf{inp}$ and $\mathbf{rdp}$, which return $\mathbf{null}$ if no matching tuple exists in the tuple space. Some variants of Linda (e.g., [17]) also provide the *bulk operations* $\mathbf{ing}$ and $\mathbf{rdg}$, which can be used to retrieve all matching tuples at once.

Processes interact by inserting tuples into the tuple space with the $\mathbf{out}$ operation and issuing $\mathbf{rd}$ and $\mathbf{in}$ operations to read and remove data from the space. A typical example is a producer/consumer, where the producer *outs* tuples describing jobs, and the consumer *ins* job tuples based on patterns related to their capabilities. If needed, the results of the job execution can be *outted* by the consumers and collected by any process with the $\mathbf{in}$ operation.

LIME*: Linda in a Mobile Environment.* Communication in Linda is decoupled in *time* and *space*, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their identity or location is not necessary for data exchange. This decoupling makes the model ideal for the mobile ad hoc environment where the parties involved in communication change dynamically due to their movement through space. At the same time, however, the global nature of the tuple space cannot be maintained in such an environment, i.e., there is no single location to place the tuple space so that all mobile components can access it at all times.

To support mobility, the LIME [13, 15] model breaks up the Linda tuple space into multiple tuple spaces each permanently attached to a mobile component, and defines rules for the sharing of their content when components are able to communicate. In a sense, the static global tuple space of Linda is reshaped by LIME into one that is dynamically changing according to connectivity. For example, consider a group of professors carrying PDAs, and imagine each of them inserting a business card tuple into their local tuple space, referred to in LIME as the *Interface Tuple Space* (ITS). When all professors are in the same room (or, according to LIME rules, within transitive communication), LIME's transient sharing of tuple spaces provides a view where it is as if all business card tuples were in the same tuple space, and accessible to all professors. However, when one professor leaves the room her business card is no longer accessible to the others, but it remains accessible to her. As shown in Figure 1, the LIME model encompasses mobile software agents and physical mobile hosts. Agents are permanently assigned an ITS, which is brought along during migration, and reside on the mobile hosts. Co-located agents are considered connected. The union of all the tuple spaces, based on connectivity, yields a dynamically changing *federated tuple space*. Hereafter, for the purpose of this work we always consider the agents as stationary.

Access to the federated tuple space remains very similar to Linda, with each agent issuing Linda operations on its own ITS. The semantics of the operations, however, is as if they were executed over a single tuple space containing the tuples of all connected components. In the previous ex-

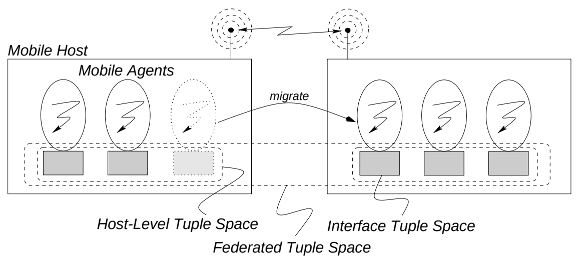

Figure 1. In LIME connected mobile hosts transiently share the tuple spaces of the agents executing on them.

ample, a professor could issue a **rd** operation and retrieve, non-deterministically, the business card of one of the professors in the room.

Besides transient sharing, LIME adds two new notions to Linda: tuple locations and reactions. Although tuples are accessible to all connected agents, they only exist at a single point in the system, i.e., with one of the agents. When a tuple is output by an agent it remains in the corresponding ITS, and the tuple location reflects this. LIME also allows for tuples to be shipped to another agent by extending the **out** operation to include a destination. The notion of location is also used to restrict the scope of the **rd** and **in** operations, effectively issuing the operation only over the portion of the federated tuple space owned by a given agent or residing on a given host. For example, a professor can read the business card of another by specifying the host identifier of her colleague as part of the **rd** query.

Reactions allow an agent to register a code fragment—a listener—to be executed whenever a tuple matching a particular pattern is found anywhere in the federated tuple space. This is particularly useful in the highly dynamic mobile environment where the set of connected components changes frequently. Continuing the example above, now a professor (say, Dr. Doe) registers a reaction for business card tuples and associates a listener for displaying the card contents. When the reaction is registered, it fires immediately for each professor, since the business cards are already in the tuple space, and trigger the display of each card content on Dr. Doe's screen. Similarly, if a new professor walks in the room with a card in her tuple space, the reaction would immediately cause its display on Dr. Doe's screen. Like queries, reactions can also be restricted in scope to a particular host or agent. Nevertheless, the ability to monitor changes across the whole system by installing reactions on the federated tuple space has been shown to be one of the most useful features of LIME.

Additional information, including API documentation and source code, is available at [1].

2.2. Crossbow Mote Sensor Platform

From this point forward our discussion focuses on the Crossbow Mote sensor platform [2], which we chose as the implementation target. It is worth noting, however, that the TinyLIME model is equally applicable to any platform. In the Crossbow platform, a sensor board can be plugged onto each mote to support several environment readings including light, acceleration, humidity, magnetic field, sound, etc. The MICA2 motes in our testbed run on two AA batteries, whose lifetime is dependent on the use of the communication and computation resources. The communication range varies greatly based on the environmental conditions, with an average indoor range observed during our experiments of 6-7 m. The motes run an open source operating system called TinyOS [7] and have 5Mbit of flash memory, with 1Mbit reserved for program memory and 4Mbit available for user data. A laptop is converted to a base station able to communicate with the motes by connecting a special base station circuit board to the serial port of the laptop.

3. Operational Setting

Most middleware designed specifically for sensor networks operate in a setting where the sensors are fixed in the environment and report their values to a centralized point. As we discuss in Section 7, much work has gone into making these operations power efficient. However, this centralized model may not be the ideal model for all applications. Consider an application that requires information from sensors in close proximity to the user. In this case, both the location of the user and the location of the sensors must be known, the data must be requested by the sensors determined as proximate, then the data must be shipped to the central collection point. This has a number of drawbacks. First, it may not be reasonable to expect to know the location of all sensors. Second, the collection of information puts a communication burden on the sensors between the proximate sensors and the collection point to forward the data. Third, it requires that all sensors be transitively connected to the base station—something that may not be feasible in all environments due to physical barriers or economic restrictions limiting the number of sensors in a given area.

Considering these issues, we propose an alternative, novel operational scenario; one that naturally provides contextual information, does not require multi-hop communication among sensors, and places reasonable computation and communication demands on the motes. The scenario, depicted in Figure 2, assumes that motes are distributed sparsely throughout a region, and need not be able to communicate with one another. The monitoring application is deployed on a set of mobile hosts, interconnected through ad hoc wireless links—e.g., 802.11 in our ex-

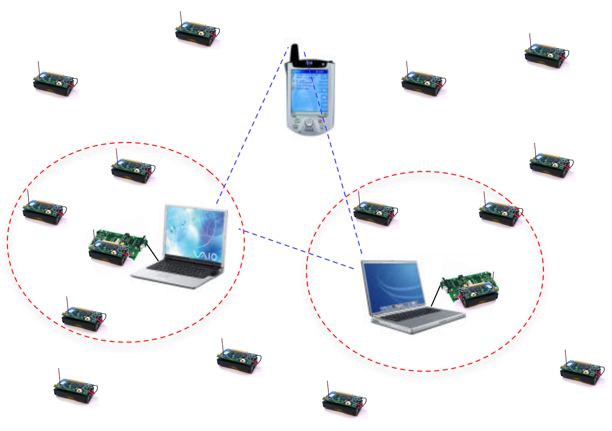

Figure 2. Operational scenario showing one hop communication between base stations (laptops) and motes and multi-hop communication among base stations and clients (PDAs). Client agents can also be co-located with the base stations (e.g., running on the laptops).

periments. Some hosts are only clients, without direct access to sensors, such as the PDA in the figure. The others are equipped with a sensor base station, which however enables access only to sensors within one hop, therefore naturally providing a contextual view of the sensor sub-system.

This scenario is not just an academic exercise, rather it is relevant in several real-world situations. Imagine, for example, a disaster recovery scenario with mobile managers and workers, where sensors have been deployed randomly (e.g., thrown from the air). The computer of each worker is also a base station that enables them to perform their tasks by accessing the proximate sensor information. Instead, managers are connected only as clients, and gather global- or worker-specific data to direct the recovery operations, without the need either to have a single data collection point or to know exactly where sensors are placed. In a sense, the scenario we propose merges the flexibility of MANETs with the new capabilities of sensor networks, by keeping the complexity of routing and disseminating the sensed information on the former, and exploiting the latter as much as possible only for sensing environmental properties.

4. The TinyLIME Model

TinyLIME was conceived to support the development of applications in the operational setting just described. It extends LIME by providing features and middleware components specialized for sensor networks. In this section we introduce the underlying model, while in the next we describe the middleware architecture.

As in LIME, the core abstraction of TinyLIME is that of a transiently shared tuple space, which in our case stores tuples containing the sensed data. However, TinyLIME also introduces a new component in addition to agents and hosts—the motes. In the physical world defined by our setting, motes are scattered around. They communicate with base stations *only* when the latter move within range. Dealing with this scenario by considering a mote just like another host would lead to a complicated model with an inefficient implementation. Instead, in TinyLIME a simpler abstraction is provided. A mote is not visible through TinyLIME unless it is connected to some base station. When this is the case, the mote is represented in the model much like any other *agent* residing on the base station host (and therefore "connected" to it), with its ITS containing the set of data provided by its sensors. Looking at Figure 1, it is as if on each host there were an additional agent for each mote currently in range of that host. Clearly, things are quite different in practice: the mote is not physically on the base station, and there is no ITS physically deployed on the mote. As usual, it is the middleware that takes care of creating this abstraction to simplify the programmer's task. The support for the abstraction is described in the next section.

Once this leap is made, the rest comes naturally. For instance, operations on the federated space now span not only connected hosts and agents, but also the motes within range of some host—similarly for operations restricted to a given host. Also, the mote identifier can be used much like an agent identifier to restrict the scope of a query or reaction to a specific mote. To make a concrete example, consider the scenario of Figure 2. When an agent on the laptop base station on the right issues a **rd** for light data, restricted in scope to its own host, a light reading from one of the two connected motes will be returned. If, instead, an agent running on the PDA client issues the same **rd** query but with unrestricted scope, a light reading from any of the five sensors connected to the two base stations will be returned nondeterministically. It should be noted, however, that although TinyLIME agents use the basic LIME operations to access sensor data, this data is read-only, i.e., only reactions, **rd**, **rdp**, and **rdg** are available. Indeed, sensors measure and report properties of the environment that cannot be changed or removed by the clients, but only inspected.

Reactions work as in LIME, modulo the changes above, and are extremely useful in this environment. Imagine a situation where a single base station agent registers a reaction to display temperature values. As the base station moves across the region, the temperature from each mote that comes into range will be displayed—with no extra effort for the programmer. TinyLIME reactions also provide additional expressive power. First, it can be specified how *frequently* the data received through a reaction should be refreshed. The notion of data freshness is a property of a sensor in the context of an application, and it reflects the fact that sensors measure environmental properties, and every reading has a time threshold beyond which it is considered no longer useful—or *fresh*. In our example, this is very useful when the base station remains in a single location for an extended period of time. Second, TinyLIME reactions also accept a *condition*, e.g., to specify reaction only to temperatures between 20 and 30 degrees. This is motivated by the need to limit communication by the motes, enabling data sending only if it is useful for the application. Both these features, together with details such as the format of sensor data tuples, are described in the next section.

5. The TinyLIME Middleware Architecture

As previously stated, the TinyLIME model has been designed and implemented for the Crossbow Mote platform, exploiting the functionalities of TinyOS. On standard hosts, TinyLIME is implemented as a layer on top of LIME without requiring any modification to it[1], therefore reasserting the versatility of the LIME model and middleware. In this section we describe the architecture of TinyLIME, whose main components are shown in Figure 3. Our discussion starts from the perspective of the client, and moves progressively towards the components deployed on the sensors.

5.1. Client Components

A client interacts with the sensors through the `MoteLimeTupleSpace` class, which extends `LimeTupleSpace` in the LIME API. However, as mentioned in Section 4, only the read and reaction operations are available: the others throw an exception. Figure 4 shows the code of a TinyLIME client agent, where the first line of the `run` method contains the creation of the `MoteLimeTupleSpace`, and the second makes it shared, as per the LIME API.

To query the motes, a client must know the format of the tuples containing sensor data. In LIME, and in general in tuple space models, this decision is normally left to the application. However, in TinyLIME this format is predefined to access the sensors available on the motes, although it can be easily adapted to suit different sensor platforms. A tuple (or template) containing a sensor reading consists of four fields: $\langle \text{SensorType}, \text{Integer}, \text{Integer}, \text{Date} \rangle$. The first field indicates the type of sensor to be queried. In our mote-specific implementation, valid values are `ACCELX`, `ACCELY`, `HUMIDITY`, `LIGHT`, `MAGNOMETER`, `MICROPHONE`, `RADIOSTRENGTH`, `TEMPERATURE`, and `VOLTAGE`. The second field contains the actual sensor reading[2]. The third field is the *epoch* number of the mote that provided the reading, and indicates approximately how long that mote has been alive, as discussed later in this section. The last field is a timestamp set when the data is collected at the base station, thus allowing correlation of values gathered at the same host without requiring synchronized clocks on the motes.

Based on this format, a client can create a `MoteLimeTemplate` for use in any of the allowed operations. The available constructors optionally allow setting a freshness value and an operation scope. The latter can span the federated tuple space, the agents or motes on a host, a single agent, or a single mote. The `MoteLimeTemplate` created in Figure 4 for a light reading specifies neither. Therefore, the **rd** using it will return as soon as a light reading with the default freshness is found in the federated tuple space. Reactions are specified as in LIME, with a template and a listener. The template, however, may contain an additional field requiring a matching other than the typical value equality provided by LIME and Linda-based models. In the current implementation, inequality (e.g., voltage different from 2.1V) and matching over a value range (e.g., temperature between 20 and 30 degrees) are available[3].

`MoteLimeTupleSpace` also provides operations dedicated to controlling sensors. For example, the PDA in Figure 2 can invoke `setBuzzer()` to cause all motes connected to the two base stations to buzz for a short period of time. Also, `setDutyCycle` changes the awake period of the motes, `setRadioPower` changes the signal strength, and `setSensingTimeout` changes the maximum time waited for a mote to answer before declaring it unreachable. Like other operations, these methods can also be restricted in scope.

5.2. Interaction Between Client and Base Station

`MoteLimeTupleSpace`, `MoteLimeTuple`, and `MoteLimeTemplate` are the only classes needed by a client application. Hereafter, we look at the internals of TinyLIME, describing how it uses LIME and interfaces with motes.

The first component we examine in detail is the `MoteLimeTupleSpace` itself. Although it presents to the client the illusion of a single tuple space containing sen-

1 For full disclosure, only a few methods changed their access level from private to protected.

2 In TinyOS all sensor readings are represented as integers. Conversion functions exist to convert them to more meaningful measurements.

3 This is only a temporary solution to enable experimentation. A new version of the tuple space engine underlying LIME [14] will provide these features as part of the template specification, therefore allowing for a more elegant and uniform solution.

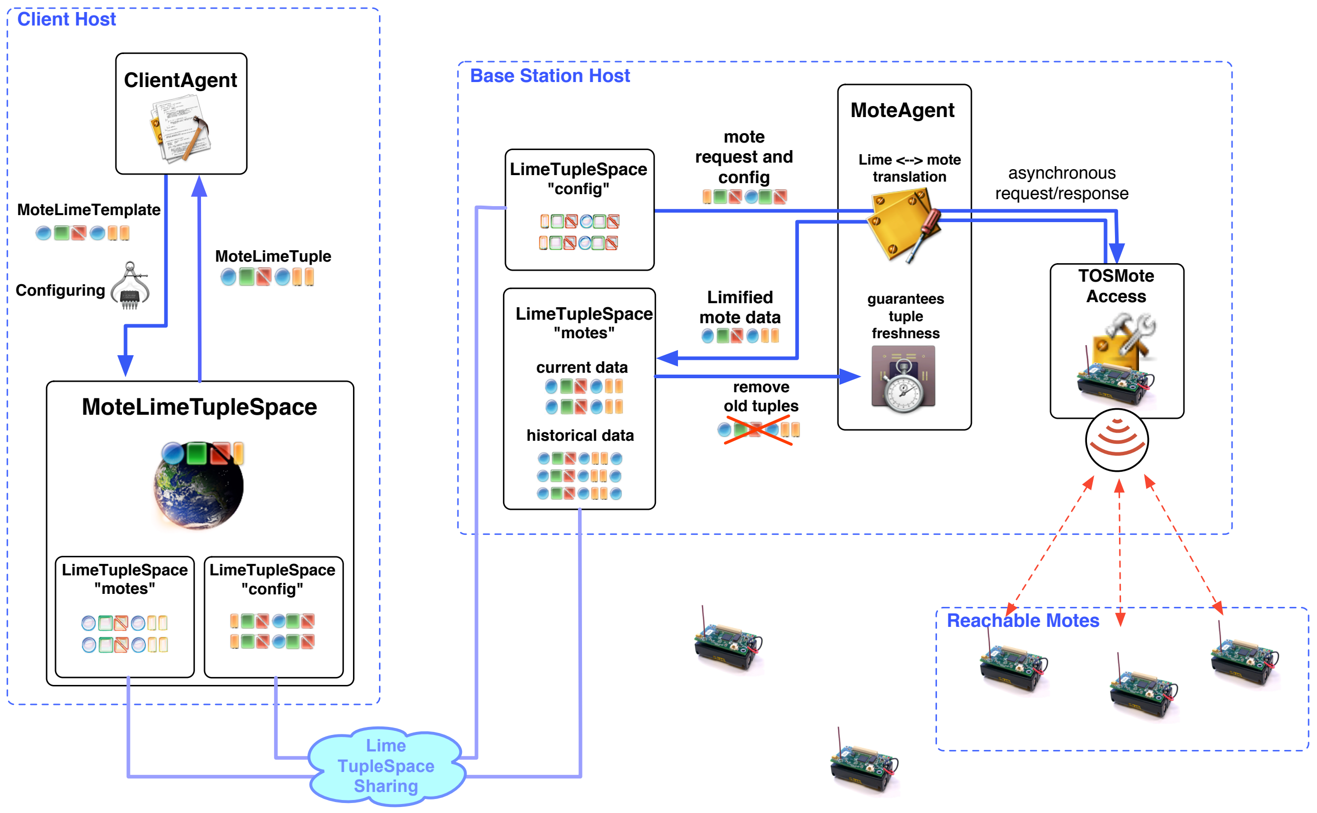

Figure 3. Main architectural components on the base station and client hosts. Although shown separate here, the two can be co-located.

```
public class SimpleClientAgent extends StationaryAgent {
  public void run() {
    LimeTupleSpace  lts = new MotesLimeTupleSpace();
    lts.setShared(true);
    ITuple tup = new Tuple()
               .addActual(new SensorType(SensorType.LIGHT))
               .addFormal(Integer.class)
               .addFormal(Integer.class)
               .addFormal(Date.class);
    MoteLimeTemplate tmpl  = new MoteLimeTemplate(tup);
    MoteLimeTuple t = (MoteLimeTuple)lts.rd(tmpl);
    System.out.println("Tuple returned: " + t.toString());
  }
}
```

Figure 4. A sample TinyLime client that reads a light sensor value and prints it to the screen. This is the real code: only exception blocks are not shown for readability.

sor data, internally it exploits two LIME tuple spaces, one holding data from the sensors and one for communicating requests from the client. By using two names, respectively *motes* and *config*, the content of the two tuple spaces is shared separately by LIME. These same two tuple spaces are also instantiated at all base stations, therefore sharing occurs across all clients and base stations, based on connectivity.

The *motes* tuple space provides access to sensor data. One would expect that, if the mote is connected, its sensor data should be in the tuple space. Instead, sensor data is retrieved only on demand, saving motes the communication of values that no application needs. Therefore, when a client issues a request, the internal processing of `MoteLimeTupleSpace` first queries the *motes* tuple space for a match. If no match is found, the operation proceeds by informing the base stations to query for the required data. This is accomplished by placing a *query tuple* into the *config* tuple space, and simultaneously registering a reaction on the *motes* tuple space. (These tuples and reactions are clearly system-defined.) The query tuple causes the firing of a reaction on the base station, which in turn retrieves the data from the sensor and posts it in the *motes* tuple space, where it causes the firing of the previously registered reaction, and delivers the data to the client. The data remains in the *motes* tuple space, possibly fulfilling subsequent queries, until it is no longer *fresh*. The freshness requirement is maintained by simply deleting the stale tuples upon expiration of a timer. The *config* tuple space is also used for implementing the mote configuration requests described in the previous section (e.g., `setRadioPower`) using a similar scheme based on request tuples and reactions.

In TinyLIME, all base stations run an instance of `MoteAgent`, which installs the system reactions necessary to the processing we described, manages the operation requests, and maintains the freshness of the sensor data. Because some applications may find it useful to access not only the current value of a sensor but also its recent values, the `MoteAgent` also maintains historical information in the *motes* tuple space, albeit with a different tuple pattern.

5.3. Base Station to Mote Interaction

To this point we have described how data is retrieved by the client once it is available, however we have not discussed in detail how it is retrieved from the motes. In TinyLIME, this is handled by a combination of three components: the `MoteAgent` that receives client requests, the `TOSMoteAccess` component that asynchronously interacts with `MoteAgent` to handle request, replies, and all communication with the motes, and finally the components residing on the motes themselves. `MoteAgent` and `TOSMoteAccess` are highly decoupled, thus enabling the reuse of the latter in applications other than TinyLIME to provide a straightforward interface to access motes from a base station.

The main job of the `TOSMoteAccess` component is to translate high-level requests issued by TinyLIME into packets understandable by the motes. Four kinds of requests are accepted: read, reaction, stop operation, and set parameter. Requests which last an extended period of time, i.e. reads and reactions, accept a listener parameter. The listener is called when the operation is complete, e.g. data is received or the timeout expires. Once a request is received by the `TOSMoteAccess` component, it is translated into communication with the motes.

Communication. In principle, communication between the base station to and from motes is just message passing. However this is not as straightforward in sensor networks as in traditional ones. To see why, one must understand a fundamental property of motes, namely that to conserve energy they sleep most of the time, waking up on a regular basis to receive and process information. Because motes cannot receive packets while sleeping, the base station must repeatedly send a single packet as shown in Figure 5. The frequency at which to repeat the packet and the length of time to repeat it are determined by two parameters: the nominal awake time and the epoch period. The *nominal awake time* is the amount of time that a mote promises to be awake during each epoch period. The *epoch period* is the basic cycle time of a mote. Multiplying the current epoch number by the epoch length estimates the time a mote has been active. To avoid duplicate delivery of packets, each contains a sequence number that the motes use to filter incoming messages. This design intentionally puts the burden of communication on the base station rather than on the motes, forcing the former to repeat a message many times to ensure its delivery. This is not an issue if, as we assume, the base station has a larger energy reserve and is more easily rechargeable than the motes.

Operation Processing. With this understanding of communication with motes, we return to the `TOSMoteAccess` component. Sending a read or parameter set request to a mote is accomplished by this component simply sending the message and waiting for the reply. The processing inside the motes will be discussed shortly, however it should be noted that because the base station does not enter a sleep mode, messages sent by the motes are only transmitted once. If no mote replies within an epoch period, the request is retransmitted. Even if no mote within range can provide the data, the base station may move into range of a new mote at any time. Therefore a **rd** request should be retransmitted as long as the client is still waiting for a tuple. The probe and group operations, **rdp** and **rdg**, must be handled dif-

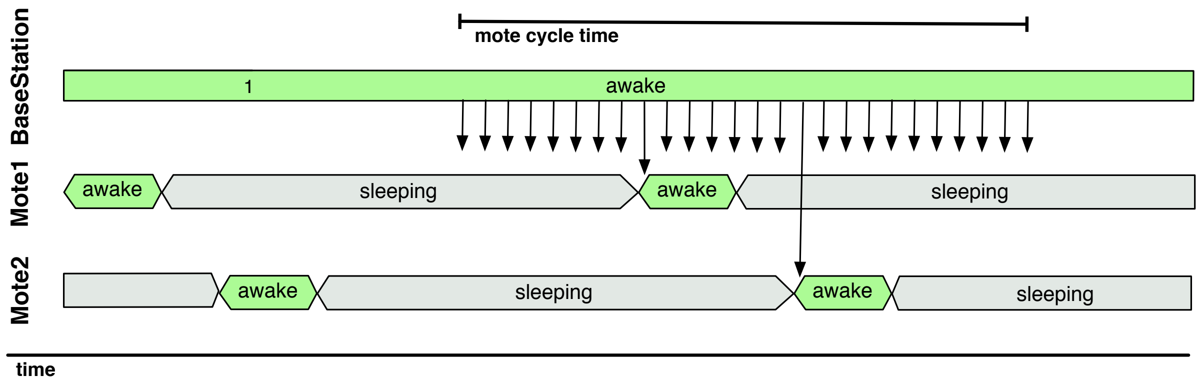

Figure 5. Communication with motes that sleep for a majority of the time. To ensure all motes in one-hop range receive the packet, each packet is broadcast by the base station at a frequency determined by the awake time for the duration of an epoch.

ferently because the agent should receive a **null** reply if no mote can service the request. Therefore, after a timeout period the `TOSMoteAccess` component stops repeating the request and returns **null** if no motes responded or, in the case of **rdg**, the set of sensor values collected before the timeout.

Next we consider reaction requests. One option is to send a reaction installation message to the motes, indicating that sensor values should be sent every epoch until they are no longer needed. This would require the motes to maintain information about all requests, including the conditions that must be met (e.g., required value ranges). At least two problems arise with this solution. Consider the case where a client/base station pair moves through a region with a reaction installed for temperature sensors. Here the base station must repeatedly send the request to install the reaction on all motes and, moreover, the motes that are no longer in contact need to detect disconnection and stop transmitting temperature values. This can be difficult and energy consuming to the motes. These problems led us to a base station driven solution, where the `TOSMoteAccess` component continuously sends reaction requests to the motes. Reaction requests differ from normal read requests because they contain the condition to be met by data, allowing the motes to avoid transmitting sensor values that are useless for the application. Motes are expected to reply once per epoch, even if their sensor value has not changed, therefore the packet filter mechanism on the motes is designed to accept packets with the same reaction request identifier once per epoch. In this solution, the motes remain stateless; when the base station moves out of range no processing is needed on the motes to cancel the operation. When a client moves out of range of the base station, the client's reactions must be disabled, but such disconnection is easily detected using LIME mechanisms inside the `MoteAgent` and the `TOSMoteAccess` is informed to stop requesting sensor values on behalf of the disconnected client. By choosing this solution, we require that the base stations both maintain more state and send more messages to repeatedly request information. However, this is reasonable given that the base station is likely to have both a larger on-board energy supply and more memory than the motes.

5.4. On-Mote Components

The only remaining component is that deployed on the motes themselves. Given the choices made up to this point, the motes component is designed as a reactive system, responding to incoming messages and of course managing its epoch and awake periods. Figure 6 shows the architecture of software deployed on motes as a set of interconnected TinyOS components. The `Timers` module controls the epoch and awake periods. The `Filtered Communication` module receives all incoming packets, eliminating duplicates based on packet identifiers. The `Sensors Subsystem` invokes the appropriate TinyOS components to take sensor readings, including powering sensors on and off before and after use. If additional sensors are added to the system, this component must be modified. The `Tuning` module handles the setting of mote parameters such as transmission power. Finally, the `Core` module links all these components together, triggering events and calling parameters.

To understand the functionality of the sensor component, consider the processing of an incoming reaction request. Assuming the incoming packet is not a duplicate, the `Filtered Communication` module passes it to the `Core`, where the sensor type and condition are extracted

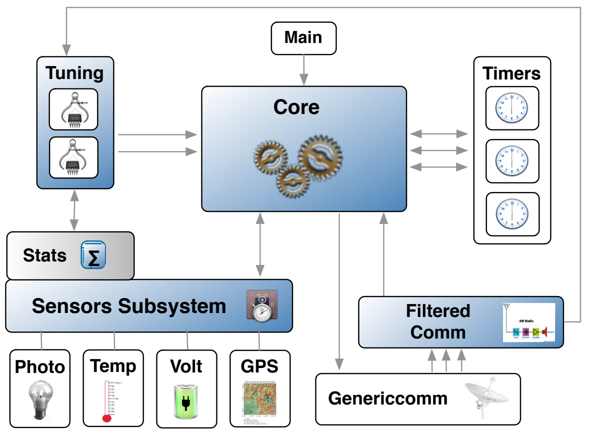

Figure 6. Architecture of components installed on motes. Shaded components have been developed for TinyLIME, while the others are provided by TinyOS.

from the packet. The `Core` then communicates with the `Sensor Subsystem` requesting a reading for the specified sensor, e.g., light. When the reading has been taken, an event is raised on the `Core`, the value is checked against the conditions contained in the packet and, if the value meets the condition, a packet containing the sensed value is assembled and passed to the `Generic Communication` module to be sent back to the base station.

5.5. An Example

To summarize how the components fit together, we walk through a simple example where a client reads a sensor light value, as shown in Figure 7. The client first creates the desired template, and invokes the **rdp** function on an instance of `MoteLimeTupleSpace`. Inside it, the **rdp** is converted to a query which is posted to the *motes* tuple space to see if a fresh light value already exists. If a value is returned, it is passed back to the client immediately. Otherwise, a configuration tuple is output to *config*, indicating that a sensed light value is needed, and at the same time a reaction is installed on *motes* for the required sensor data. The `MoteAgent`, which is registered to react to every configuration tuple, receives the agent's request, passes it along to the `TOSMoteAccess` component, which in turn sends a read request to the motes. When the value is returned, it is placed into the *motes* tuple space, triggering the earlier installed reaction to fire, which finally delivers the tuple with the sensed value back to the client.

Component	Language	LOC
`MoteLimeTupleSpace`	Java	446
`MoteAgent`	Java	361
`TOSMoteAccess`	Java	470
On-Mote Components	nesC	550

Table 1. Uncommented lines of source code for all major TinyLIME components.

6. Implementation Details

TinyLIME is available for download at `http://lime.sf.net/tinyLime.html`. Table 1 shows the breakdown of source code lines across components.

Evaluation. To get a feel for response times, we ran some indicative test cases with an epoch time of $8s$ and an awake time of $2s$. These values, especially the epoch time, heavily influence the numbers below, and should be tuned according to the application needs.

Our test cases involved only the blocking **rd** operation. Results would have been the same for the **rdp** since in our test a mote is always in range to provide the requested value. The **rdg** instead would produce different but rather uninteresting results, because its performance is not dependent on the motes but on the parameter of the `TOSMoteAccess` component that determines how long to wait to collect all replies. All tests were run with the TinyLIME client colocated with the base station in order to eliminate the network delays on the non-mote network.

Our first test involved a single mote and two requests. For the first **rd** request, response times varied from $0.35s$ to $5.8s$, with an average over 11 runs of $3.2s$. In a twelfth run, the response time was observed at $12.1s$, clearly an unexpected value since it is longer than the epoch time. This can be explained by the lossy nature of mote communication. Likely, the request packet was corrupted and the mote did not receive the request until the second epoch. Immediately following the first **rd** request, a second request was issued. In this case, the previously sensed value was still considered fresh, so no communication with the motes was required. This time, response times varied from $0.0049s$ to $0.20s$, depending on CPU load. With low CPU usage, the average was approximately $0.008s$.

Our second set of tests involved three motes. In this case, the first **rd** request response times varied from $0.29s$ to $2.3s$ with an average of $1.2s$. This reduction in time over the single mote scenario is expected since the awake time of the motes is likely to be scattered, increasing the chance that at least one of the motes is awake shortly after the query is issued. We also repeated the test with a second, immediate **rd** showing the same results as before. Again, this is expected

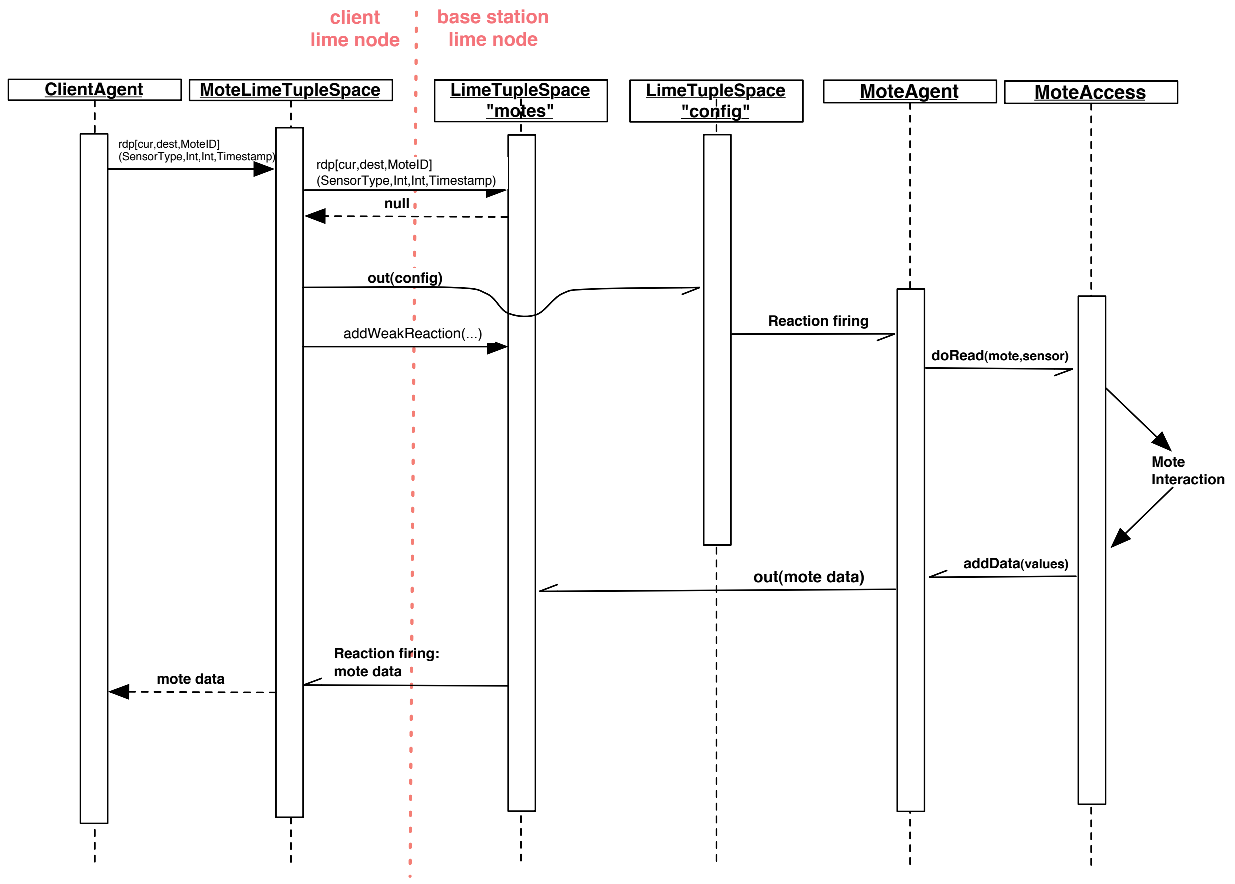

Figure 7. Sequence diagram showing the processing of a rdp**.**

since no mote communication is involved as the fresh value is simply observed in the tuple space.

Tests with reactions confirm the previous results, assuming the readings match the specified conditions. Again, this is expected since reaction are implemented like queries—they are simply repeated for more than one epoch.

Wakeup scattering. As observed during the tests, the response time for a request is quite variable and dependent on when it is issued with respect to when the motes wake up and receive the request packet. With multiple motes the average response time was shorter, albeit still quite variable. This is because the awake times of the motes are not coordinated in any fashion; motes wake up at random times in the epoch period and these times are not evenly distributed. The left side of Figure 8 shows the consequence of this phenomenon. If the first transmission of the request is at A, then the response will be immediate. However, if the first transmission is at B, no mote is awake to reply, and the request will not be answered until the second transmission during the next mote epoch. Interestingly, even if the first transmission is at A and TinyOS provides provisions such as carrier sense and collision avoidance for multiple motes transmitting at the same time, multiple motes can still saturate the channel and affect performance. Again the left half of Figure 8 shows this possibility with four motes, whose wakeup times are not exactly aligned, but nonetheless may respond all at the same time when a request is made.

To avoid these situations, we propose a wakeup scattering algorithm that more uniformly distributes the points at which the motes wake up during the epoch, resulting in a wakeup distribution similar to the right side of the figure. Although a centralized solution may be able to optimize a large, multi-hop network, our solution is distributed, and enables local decisions with relatively little overhead. In short, our solution proceeds in rounds. At the beginning of the first round, the process is initiated by a special packet sent by one of the motes or an external agent. Immediately after receiving this packet, each mote randomly places its wakeup time in the next one-epoch-long period of time (the calibration period). When this time arrives, it sends a *scatter notify* packet. During the entire epoch, each mote records the arrival times of the notify packets of the others. Only two such packets are of interests, the one received immediately be-

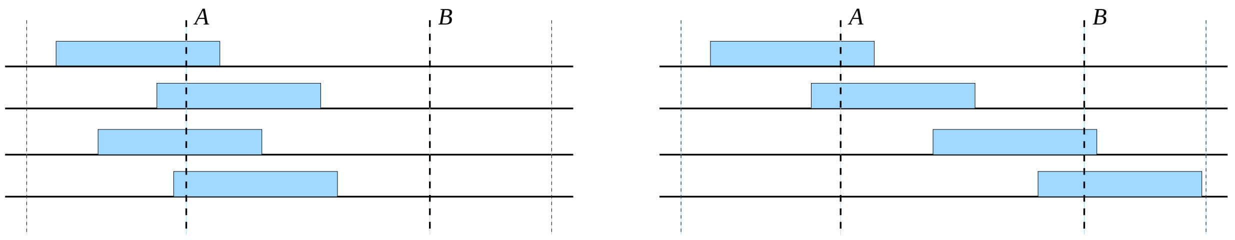

Figure 8. Left: with non-ideal wakeup times, all motes compete for bandwidth to reply to the query and there may be a delay if the first transmission is at time B. After scatter wakeup times, fewer motes compete and with higher probability some mote is awake to receive the transmission.

fore the transmission of the mote's own scatter notify, and the one received immediately after. At the end of the epoch, the mote finds the midpoint between these two transmissions, and moves its own wakeup time closer to, but not exactly to, this point. How far to move the wakeup time is a parameter we determined through experimentation. The process of detecting the scatter notify packets and moving the wakeup time can be repeated any number of times, iteratively refining distribution.

This solution has been implemented directly in nesC and simulated in tossim, a TinyOS simulator. It yielded the expected results, namely distributing the wakeup times more evenly throughout the epoch. We are still evaluating several aspects of this approach including choosing how far to move the wakeup time each epoch, evaluating how many rounds are needed on average to stabilize the wakeup time, and how the algorithm is affected by the fact that the initialization packet will not reach all motes at the same time.

7. Related Work

The idea of providing middleware for sensor networks has been growing in popularity, providing application programmers with a variety of useful abstractions easing the development process. EnviroTrack [3], a middleware for environmental tracking applications, supports event-driven programming by identifying an event at a given location, collecting the data from proximate sensors, and reporting the readings and event to the user. TinyLIME supports a similar notion through reactions, although it does not perform the in-network aggregation of EnviroTrack.

An alternative model is data-oriented and thus closer to TinyLIME. In Directed Diffusion [8], applications specify "interest queries" for the necessary data attributes, and the nodes collaborate to set up routes for this information to follow back to the application. It is explicitly multi-hop in nature, unlike TinyLIME that focuses on local, contextual interactions with sensors. Other systems provide database interaction with sensors. TinyDB [12] provides an SQL-like interface with optimization for placement of parts of the query (e.g., joins, selects) to minimize power consumption. Cougar [4] and SINA [18] also provide a distributed database query interface towards a sensor network with an emphasis on power management either by distributing queries or clustering low-level information in the network. Although TinyLIME also provides a simplified database model, the Linda tuple space, it has no notion of collecting information at a single point. Instead multiple clients can be distributed, and the system can dynamically reconfigure, something not inherent in the other systems. Moreover, since TinyLIME protocols are simpler and do not require a tree structure, its communication delays tend to be smaller.

Other data-oriented approaches, such as DSWare [11], address the redundancy of data collected by geographically proximate sensors. By aggregating the data of several sensors and reporting it as a single value, some amount of sensor failure can be tolerated. QUASAR [10] addresses quality concerns, allowing applications to express Quality aware Queries (QaQ). For example, QaQs can express quality requirements as either set-based (e.g., find at least 90% of the sensors with temperature greater than 50^oC) or value-based (e.g., estimate the average temperature within 1^oC). Neither of these optimizations is incorporated into TinyLIME, although we plan to explore how aggregation and quality concerns can be addressed in the system.

Tuple spaces have also been considered previously for use in sensor networks. Claustrophobia [5] replicates a single tuple space among multiple motes, providing a variety of efficiency-reliability tradeoffs for populating the tuple space with sensor data as well as retrieving that data. However, it is based on a different operational setting than the one we chose in this paper. ContextShadow [9] exploits multiple tuple spaces, each holding only local information thus providing contextual information. The application is required to explicitly connect with the tuple space of interest to retrieve information. TinyLIME, being focused on the combination of MANET and sensor networks, exploits physical locality to restrict interactions without application intervention.

8. Conclusions

In this paper we proposed a novel operational setting for wireless sensor networks, and a middleware supporting the development of corresponding applications. The operational setting, in contrast with mainstream approaches, does not assume a centralized data collector. Data is cooperatively collected by mobile monitors interconnected through a MANET, which can access only those sensors that are directly available to them. This configuration brings more flexibility, simplifies the communication infrastructure, and naturally provides for context-awareness through proximity of sensors to the monitors. The middleware, TinyLIME, is an extension of the LIME middleware originally designed for MANETs. While communication between the mobile monitors is entirely handled through LIME, a new layer is built entirely on top of it to interact efficiently with the specialized components deployed on the sensors.

An instantiation of the middleware has been implemented for the Crossbow mote platform, and is available for download at `http://lime.sf.net/tinyLime.html`.

References

[1] Lime. `http://lime.sourceforge.net`.

[2] Crossbow Technology Inc. `http://www.xbow.com`, 2005.

[3] T. Abdelzaher, B. Blum, D. Evans, J. George, S. George, L. Gu, T. He, C. Huang, P. Nagaraddi, S. Son, P. Sorokin, J. Stankovic, and A. Wood. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proc. of the 24th Int. Conf. on Distributed Computing Systems (ICDCS)*, 2004.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Querying the physical world. *IEEE Personal Communication*, 7(5):10–15, October 2000.

[5] V. Bychkovskiy and T. Stathopoulos. Claustrophobia: Tiny tuple spaces for embedded sensors. Course Project, UCLA CS233, `http://lecs.cs.ucla.edu/~thanos/cs233/cs233_vlad_thanos_report.ps`, 2002.

[6] D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, January 1985.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, USA, November 2000.

[8] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of the 6th Int. Conf. on Mobile Computing and Networks (MobiCom)*, 2000.

[9] M. Jonsson. Supporting context awareness with the context shadow infrastructure. In *Workshop on Affordable Wireless Services and Infrastructure*, June 2003.

[10] I. Lazaridis, Q. Han, X. Yu, S. Mehrotra, N. Venkatasubramanian, D. Kalashnikov, and W. Yang. QUASAR: Quality-aware sensing architecture. *SIGMOD Record*, 33(1):26–31, March 2004.

[11] S. Li, S. Son, and J. Stankovic. Event detection services using data service middleware in distributed sensor networks. In *Proc. of the 2nd Int. Workshop on Information Processing in Sensor Networks*, April 2003.

[12] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[13] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 524–533, May 2001.

[14] G. P. Picco, D. Balzarotti, and P. Costa. LIGHTS: A lightweight, customizable tuple space supporting context-aware applications. In *Proc. of the 20th ACM Symposium on Applied Computing (SAC)*, Santa Fe, New Mexico, USA, March 2005.

[15] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering (ICSE)*, pages 368–377, May 1999.

[16] J. Polastre, R. Szewcyk, A. Mainwaring, D. Culler, and J. Anderson. Analysis of wireless sensor networks for habitat monitoring. In Raghavendra, Sivalingam, and Znati, editors, *Wireless Sensor Networks*, pages 399–423. Kluwer Academic Pub, 2004.

[17] A. Rowstron. WCL: A coordination language for geographically distributed agents. *World Wide Web Journal*, 1(3):167–179, 1998.

[18] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communication*, 8(4):52–59, August 2001.