Mobile Data Collection in Sensor Networks: The TinyLime Middleware

Carlo Curino [a] Matteo Giani [a] Marco Giorgetta [a]
Alessandro Giusti [a] Amy L. Murphy [b] Gian Pietro Picco [a]

[a] *Dip. di Elettronica e Informazione, Politecnico di Milano, Italy*
[b] *Dept. of Informatics, University of Lugano, Switzerland*

Abstract

In this paper we describe TinyLime, a novel middleware for wireless sensor networks that departs from the traditional setting where sensor data is collected by a central monitoring station, and enables instead multiple mobile monitoring stations to access the sensors in their proximity and share the collected data through wireless links. This intrinsically context-aware setting is demanded by applications where the sensors are sparse and possibly isolated, and where on-site, location-dependent data collection is required. An extension of the Lime middleware for mobile ad hoc networks, TinyLime makes sensor data available through a tuple space interface, providing the illusion of shared memory between applications and sensors. Data aggregation capabilities and a power-savvy architecture complete the middleware features. The paper presents the model and application programming interface of TinyLime, together with its implementation for the Crossbow MICA2 sensor platform.

Key words: wireless sensor networks, mobile ad hoc networks, middleware, tuple space

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 [Polastre et al., 2004], this is a natural setting. However, in many others this seems overly constraining. First of all, it may be impractical or even impossible to choose the location of 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, for example, reading only the average temperature sensed around a technician while he walks through a plant.

In this paper, we outline a new operational setting for sensor network applications, and describe a new middleware to support development in this arena. Our reference operational setting replaces centralized data collection with a set of mobile monitors 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. The monitors are interconnected through ad hoc, wireless links of which they can share locally collected data.

To support this operational setting, we extend and adapt a model and middleware called Lime [Picco et al., 1999, Murphy et al., 2001], originally designed for mobile ad hoc networks (MANETs). Changes to the model are required to match the operational setting and extensions of the middleware are needed to cope with the requirements related to power consumption and the sheer need of installing the middleware components on devices with very limited computational resources. Key features include operations with highly tunable scope and options for treating data in aggregated, as well as raw form. The result of this effort, called TinyLime, has been implemented entirely of top of the original Lime and deployed using Crossbow MICA2 motes [xbo, 2005] 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. Section 5 describes the features available to clients by introducing the application programming interface through sample code. Section 6 describes the internal architecture of the middleware. Additional implementation details and evaluation are provided in Section 7. Section 8 places our work in the context of related efforts. Finally, Section 9 ends the paper with brief concluding remarks.

2 Background — Linda and 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 [Gelernter, 1985] 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 **out**(t) operation and remove tuples by executing **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 **in** is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive **rd**(p) operation.

Both **in** and **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 **inp** and **rdp**, which return **null** if no matching tuple exists in the tuple space. Some variants of Linda (e.g., [Rowstron, 1998]) also provide the *bulk operations* **ing** and **rdg**, which can be used to retrieve all matching tuples at once.

Processes interact by inserting tuples into the tuple space with the **out** operation and issuing **rd** and **in** operations to read and remove data from the space.

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.

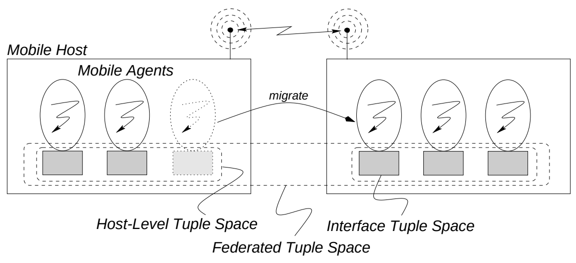

Fig. 1. In LIME connected mobile hosts transiently share the tuple spaces of the agents executing on them.

To support mobility, the LIME [Picco et al., 1999, Murphy et al., 2001] 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. 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.

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.

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. 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 [lim, 2000].

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 8, much work has gone into making these operations power efficient. However, this centralized approach may not be the appropriate 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 sensors. The scenario, depicted in Figure 2, assumes that sensors 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 experiments. 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 subsystem.

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-

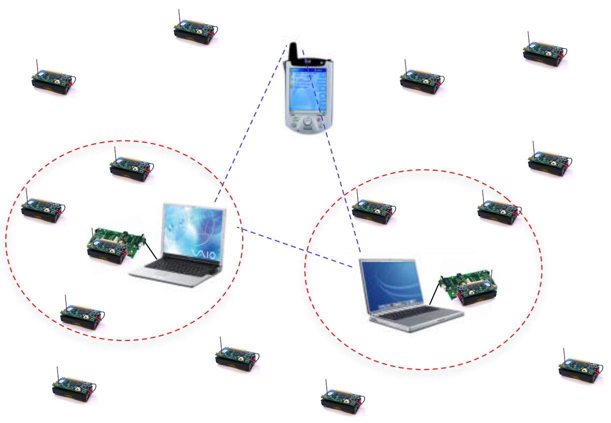

Fig. 2. Operational scenario showing one hop communication between base stations (laptops) and sensors and multi-hop communication among base stations and clients (PDAs). Client agents can also be co-located with the base stations (e.g., running on the laptops).

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 (and even workers) 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 TinyLime: An Overview

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. The implementation has been built for the Crossbow Mote sensor platform. In this section we overview this platform, introduce the model underlying TinyLime, and in general provide an overview of its main concepts and capabilities.

Crossbow Mote Sensor Platform. We selected the Crossbow MICA2 mote sensor platform [xbo, 2005], as our 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 sensor node (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 [Hill et al., 2000] and have 5Mbit of flash memory, with 1Mbit (512KB) reserved for program memory and 4Mbit (512KB) 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.

Programming Model: from Lime to TinyLime. 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 component in addition to agents and hosts—the motes. In the physical world, motes are scattered around and 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 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 following section.

The rest of the model follows 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 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 non-deterministically. 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 by accepting a *condition*, e.g., to specify reaction only to temperatures between 20 and 30 degrees. This helps to limit communication by the motes, enabling data transmission only if it is useful for the application.

Time Epochs and Data Freshness. Sensors measure environmental properties and, in many cases, the data they sense is useful only if it is *recent enough*. To empower the programmer with the ability to specify constraints about the obsolescence of the data sensed, all TinyLime operations enabling data access (i.e., proactive queries such as **rd** as well as reactions) accept a *freshness* parameter. This also enables caching on the base stations, returning matching data stored in the tuple space without communicating with the sensor nodes—and therefore saving their power—as long as the data is fresh enough.

However, to specify the freshness of data with respect to time, one needs a representation of time. Physical time requires complex synchronization algorithms [Sundararaman et al., 2005] that introduce significant communication overhead. Moreover, only a fraction of sensor network applications require the degree of precision provided by these algorithms. Instead, we divide time in intervals called *epochs*. As described in Section 6, sensing and communication is governed by the duration of the epoch, which is a system-wide deployment-time configuration parameter. In particular, data is only recorded at most once per epoch. Therefore, epochs are a natural way to express constraints over time. For instance, it is possible to specify that the data returned by a **rd** should not be older than 5 epochs. Each mote tracks the number of epochs is has been alive and associates each reading with this *epoch count*. This allows temporal comparison of readings from the same mote. Because motes have no notion of wall clock time, and each is likely to have a different epoch counter from having booted at different times, all requests for data sent to motes express time in terms of epochs relatives to *now*, e.g., 10 to 5 epochs ago.

Data Aggregation and Sensor Operation Modes. Sensor network applications often do not simply gather raw data, rather they collect and transform it, e.g., aggregating values to find the average temperature over a time interval to reduce the impact of spurious readings. Such aggregation can be applied at two different levels: globally, over the values sensed by multiple sensors, or locally, over the values sensed by a single sensor. Both aggregation modalities are supported by TinyLime.

Aggregation over multiple sensors is easily achieved using the basic TinyLime operations. For instance, the **rdg** operation can be used to collect all tuples containing light sensor readings, which can then be aggregated using appropriate application logic encoded in the agent. Instead, aggregation over the values of a single sensor requires specific middleware support. In principle it is possible to adopt the same strategy as above—collect raw values from a sensor using a **rd** and aggregate them on the base station—but this is inefficient in terms of communication bandwidth, and therefore power consumption on the sensor nodes. Whenever possible, aggregation should be pushed close to the data and therefore on the sensor nodes, trading computation for communication—a desirable strategy for sensors, where communication is much more expensive than computation and sampling [Anastasi et al., 2004].

This goal is achieved in TinyLime by making two significant extensions to what has been presented thus far. First of all, there must be a way to request access to data by specifying not only that an aggregate value should be returned, but also which aggregating function should be exploited. This is achieved essentially by exploiting an appropriate format for the templates and tuples involved in operations on the tuple space. Second, implementing aggregation on the sensor nodes entails a completely different change of perspective. Thus far, we assumed that the sensors play a *passive* role, by acquiring and communicating data only if and when needed—i.e., when the base station makes a request with a **rd** or reaction operation. Instead, aggregation requires an *active* behavior from the sensors, which must be able to autonomously and periodically sample the data over which aggregation is computed. Conceptually, it is as if sensors keep their readings in a private data space and allow access to it through an aggregating function. Nevertheless, the active and continuous sampling implied by this form of aggregation is fundamental for some applications, but may be overkill for others, where instead it is sufficient to activate sampling only when a finer-grained analysis of a phenomenon is needed—or never at all. Therefore, TinyLime gives the programmer both options. A sensor can be preprogrammed to take samples at regular intervals throughout its lifetime. Alternately, it can start in passive mode, and it is up to the application programmer to explicitly activate sampling for a given number of epochs, after which the sensor automatically switches itself back into passive mode, without additional communication. The next section describes in more detail how this and other aspects of TinyLime are made available through its application programming interface.

5 Programming with TinyLime

Before exploring the internal architecture of TinyLime, we illustrate how TinyLime can be exploited to develop applications accessing sensor data. We first present the application programming interface (API) of our middleware, and then show it in action through two simple examples, one for reading a single sensor value and the other for reacting to aggregated values.

5.1 API

In many respects, the interface to TinyLime is similar to that of Lime, presenting a tuple space interface for retrieving data. Figure 3 shows the main components of the API, specifically the tuple space class, the tuple and template classes, and the interface for specifying aggregation options. The interested reader can find the full API on the web page [tin, 2000].

The primary mechanism for a client to interact with the sensors is through the `MoteLimeTupleSpace` class, which extends `LimeTupleSpace` from the Lime API. However, in TinyLime, client applications do not create sensor data, therefore creation of tuples by clients is not allowed. Similarly the operations that remove tuples are prohibited since they would allow a client to remove sensor data, deemed part of the environment, and therefore not able to be removed. Instead, only the read and reaction operations are allowed on current sensor data. The other operations throw an exception for sensor data.

The main parameter of the read operations and reactions is the tuple template. In Lime, and in general tuple space models, the format of this template is left to the application. However, in TinyLime this format is predefined to access the sensor data available from the motes, although it can be easily adapted to suit different sensor platforms. As shown in Figure 4, TinyLime uses four specific tuple templates to access the different types of data.

A tuple (or template) for a regular sensor reading consists of four fields. 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[1]. The third field is the *epoch* number at the mote when the reading was taken. It indicates approximately how long the mote has been alive and allows temporal comparison of readings from the same mote. The last field represents information provided by the base station

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

```
public class MoteLimeTupleSpace extends LimeTupleSpace {
     public MoteLimeTupleSpace();
     LimeTuple     rd(LimeTemplate template);
     LimeTuple[]   rdg(LimeTemplate template);
     LimeTuple     rdp(LimeTemplate template);
     RegisteredReaction[]     addWeakReaction(Reaction[] reactions);
     RegisteredReaction[]     getRegisteredReactions();
     void     removeWeakReaction(RegisteredReaction[] reactions);
     // Note the special-purpose IDs:  AgentLocation.UNSPECIFIED, MoteID.UNSPECIFIED
     int      getSensingTimeout(AgentLocation baseID, MoteID mid);
     void     setSensingTimeout(AgentLocation baseID, MoteID mid, int sensingTimeout) ;
     void     setBuzzer(AgentLocation baseID, MoteID mid);
     void     setRadioPower(AgentLocation baseID, MoteID mid);
     void     setSensingTimeout(AgentLocation baseID, MoteID mid)
     void     setActive(AgentLocation baseID, MoteID mid,
                        SensorType[] sensors, int timeout);
  }

public class MoteLimeTemplate extends LimeTemplate {
     public MoteLimeTemplate(Location cur, AgentLocation dest, MoteID mote,
                int freshness, LimeTupleID id, ITuple t);
     public MoteLimeTemplate(ITuple t);
     public MoteID getMoteid();
     public MoteID setMoteid();
     public int getFreshness();
     public int setFreshness();
}

public class MoteLimeTuple extends LimeTuple {
     public MoteLimeTuple(MoteID mid, AgentLocation cur, AgentLocation dest,
                LimeTupleID id, ITuple t);
     public MoteID getMoteid();
}

public interface IAggregationOptions {
    int getFunctionID(); // type of aggregation selected
    int getEpochFrom();  // beginning epoch of aggregation range
    int getEpochTo();    // ending epoch of aggregation range
}
```

Fig. 3. TinyLime API.

Operation	Data	Template
Query	Singleton	⟨SensorType: st, Integer: sensorReading, Integer: readingEpoch, BaseStationInfo: bi⟩
	Aggregate	⟨SensorType: st, IAggregationOptions: opt, Integer: aggReading, Integer: startEpoch, Integer: endEpoch, BaseStationInfo: bi⟩
Reaction		⟨<*singleton or aggregate template fields*>, Condition: cond⟩

Fig. 4. Tuple templates used by TinyLime clients.

when the sensor value is collected. The details are left to the programmer who provides a class that decorates `BaseStationInfo`. In our tests we use the timestamp of the base station, to approximate when the value was read. Alternatives include recording any base station information such as location, remaining battery power, or the signal-to-noise ratio for communication with the mote that provided the reading.

Following this format, a client can create a `MoteLimeTemplate` for use in any of the allowed operations. The constructors optionally allow setting a freshness value and an operation scope. The latter allows spanning the federated tuple space, the agents or motes on a host, a single agent, or a single mote.

Tuples for retrieving aggregate information require the client to specify aggregation parameters. As before, the first field of the tuple identifies the sensor type to be aggregated. The next field uses the `IAggregationOptions` interface to specify the details of the aggregation including the aggregation function and the interval over which to perform the aggregation. Our implementation provides `AVERAGE`, `MAXIMUM`, `MINIMUM`, and `VARIANCE`, but this can be expanded by the programmer by implementing both the `IAggregationOptions` interface and the mote-side code to perform the function, described later. The start and end epochs for aggregation are expressed relative to the current epoch. In other words, a range of $(10, 5)$ aggregates values from 10 epochs ago until 5 epochs ago. The next integer in the template is the sensor reading. The following two integers represent the actual epoch interval the aggregation was executed over. They should be left as formal in the template, but will be bound to the absolute epoch values from the mote that did the aggregation. The final field of the aggregate data template is the `BaseStationInfo`, which is the same as in other templates.

Reactions add an additional level of complexity to the templates. Specifically the template, either for a regular data value or an aggregate value, is appended with a condition 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[2].

`MoteLimeTupleSpace` also provides operations dedicated to controlling sensors. For example, the PDA in Figure 2 can invoke:

```
setBuzzer(AgentLocation.UNSPECIFIED, MoteID.UNSPECIFIED)
```

to cause all motes connected to the two base stations to buzz for a short period of time. `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. As mentioned earlier, motes must actively record values in order for aggregation to have meaning. This is controlled using `setActive` whose parameters specify the scope of the activation and the duration of the logging in terms of epochs. Like other tuple space operations, these methods can also be executed over various scopes.

[2] This use of range matching is similar to a new feature in the tuple space engine underlying LIME [Picco et al., 2005] that provides range matching as part of the template specification. In principle, TinyLIME can be changed to exploit this, however as of this writing the integration has not been completed.

```
public class ReaderAgent extends StationaryAgent {
  public void run() {
    LimeTupleSpace  lts = new MoteLimeTupleSpace();
    lts.setShared(true);
    ITuple tup = new Tuple().addActual(new SensorType(SensorType.LIGHT))
                            .addFormal(Integer.class)   // sensor reading
                            .addFormal(Integer.class)   // epoch
                            .addFormal(BaseInfo.class); // e.g., timestamp or location
    MoteLimeTemplate tmpl = new MoteLimeTemplate(tup);
    MoteLimeTuple t = (MoteLimeTuple)lts.rd(tmpl);
    System.out.println("Tuple returned: " + t);
  }
}
```

Fig. 5. A sample TinyLime client agent that reads a light sensor value and prints it to the screen. This is actual code: only exception blocks are omitted for readability.

5.2 Reading a Sensor Value

To make the use of the API concrete, Figure 5 shows the code of a simple TinyLime client agent that reads and prints a single light reading. The code shows the creation and sharing of the tuple space, the formation of the sensor template and the retrieval of the value. The scope of the operation is set as part of the `MoteLimeTemplate` itself. By constructing the template with no specific scope options, in the example, the entire federated tuple space is queried, meaning any motes in range of any base stations connected to the client can respond. If the client is co-located with a base station, then another meaningful scope is the base station host, retrieving a light reading from one of the motes close to the client. Similarly, if a different base station identifier or a specific mote identifier are known, the client can differently restrict the scope, e.g., to explicitly request from a part of the sensed region far from the client.

Use of the blocking read, **rd**, dictates that the client will not continue until a light reading is returned. If no motes are connected that can supply the requested value, the client will block until a sensor comes into range. Non-blocking semantics can be achieved by using the **rdp** operation, which, however requires restricted scope.

To retrieve an aggregated value instead of a single reading, only the tuple template needs to change. In the following, we describe the use of the aggregation template used for a reaction.

5.3 Reacting to Locally Aggregated Sensor Values

Our experience with Lime refpicco00:developing showed that reactions are a useful programming tool and often constitute a large fraction of application code. This is even more true in the sensor environment as reactions naturally allow many useful access patterns. For example, a client to receive notification when readings cross a defined threshold, to receive values when a new mote comes within range, or simply to receive new readings once per epoch as they are generated by the motes. In this section we begin by describing in detail an example for a scenario in which a single client is resident on a single, non-moving base station. The example is simplistic for the sake of presentation: more sophisticated variants are presented at the end of the section.

Assuming the aforementioned simple scenario, the agent shown in Figure 6 requests the motes to log temperature values for the next 100 epochs. It then a creates a tuple template to request aggregated temperature values meeting a condition. The specific aggregation function requested is the average over 10 epochs (range $(10, 0)$, meaning from 10 epochs ago until the current epoch) and the condition is any aggregate whose value is greater than 200 [3] . Additionally, the aggregate returned must be over at least 5 values. If the aggregate is requested before the mote has made at least 5 temperature readings, it will not respond. This template is used to register a reaction. When this reaction fires, the `reactsTo` code at the bottom of the figure is executed, printing the identifier of the mote that returned the aggregate value and the tuple itself.

One of the simplifying elements in this example is the assumption of maximal scope for all operations, setting active all motes in the federation and reacting to all aggregates matching the criteria. Because we assume only one base station is in the system, our example has the effect of retrieving information from the motes close to the client. If, however, there were more base stations, the `setActive` operation could be restricted in scope only to the base station of the client, or could be opened to all motes connected to any of the base stations in the current federation. This allows a single client to easily compare local values against remotely collected values.

In our example, a single agent both requests and collects aggregates. An interesting option is to divide these tasks across two agents. For example, consider a scenario in which a client/base station pair moves through the field of sensors and selectively sets regions of sensors to log sensor values. Another agent is later tasked to move through the field to retrieve sensor readings. This coordination among the activator and collector can be arranged externally, or by exploiting coordination with regular Lime tuple spaces.

[3] As mentioned previously, motes return only integer values that must be converted into meaningful readings. Before this conversion, our experiments yielded room temperature of around 120.

```
public class AggregateReactionAgent extends StationaryAgent implements ReactionListener {
  public void run() {
    LimeTupleSpace  lts = new MotesLimeTupleSpace();
    lts.setShared(true);
    // set motes to record temperature readings for 100 epochs
    lts.setActive(AgentLocation.UNSPECIFIED, MoteID.UNSPECIFIED,
                  {SensorType.TEMP}, 100);

    // wait while readings are actually made so that aggregation makes sense...

    // Take the AVERAGE of the last 10 epochs, minimum of 5 epochs in returned aggregate
    IAggregationOptions aggOpt = new AggregationOptions(10, 0, AggregationOptions.AVERAGE, 5);

    // construct template to react to temperatures averaged over 10 epochs
    //  whose aggregate value is greater than 200
    ITuple t = new Tuple().addActual(SensorType.TEMP)
                          .addActual(aggOpt)
                          .addFormal(Integer.class)  // aggregated value
                          .addFormal(Integer.class)  // start epoch
                          .addFormal(Integer.class)  // end epoch
                          .addFormal(BaseInfo.class) // e.g., timestamp or location
                          .addActual(new Condition(Condition.GREATER_THAN, 200, 0));
    MoteLimeTemplate tmpl = new MoteLimeTemplate(t);

    // define and register the reaction. The listener is the agent itself
    // (see the reactsTo method definition below)
    UbiquitousReaction ur = new UbiquitousReaction(tmpl, this, Reaction.ONCEPERTUPLE);
    RegisteredReaction rr = lts.addWeakReaction(new UbiquitousReaction[]{ur});

    // client continues processing as normal...
  }

  public void reactsTo(ReactionEvent mre) {
    // Print the moteid where the value came from
    System.out.println(((MoteTuple)mre.getEventTuple()).getMoteID());
    // Print the tuple with the sensor value
    System.out.println((mre.getEventTuple()).getTuple());
  }
}
```

Fig. 6. A sample TinyLime client that reacts to temperature readings aggregated over the last 10 epochs.

5.4 *Exploiting Global Aggregation*

As a final example we consider aggregation of multiple sensor values. Unlike the local aggregation above in which the aggregation is performed on the motes, this aggregation is performed by the client. The readings to aggregate are collected with the **rdg** operation and the function itself is defined in client-side code. To change the first example to aggregate the light readings of all sensors in range of a base station co-located with the agent, only the construction of the template and the retrieval of the data must be changed as in the following:

```
tmpl = new MoteLimeTemplate(thisHost, AgentLocation.UNSPECIFIED, MoteID.UNSPECIFIED,
                            Freshness.DEFAULT, null, tup);
MoteLimeTuple[] readings = rdg(tmpl);
```

Above we use the template constructor with parameters for specifying a host identifier, restricting the scope of the operation to the host where the client and the base station reside (`thisHost`). The agent and mote identifiers are left unspecified, as are the freshness and tuple identifier. An array of tuples is returned from the **rdg** and an aggregation function, e.g., average, can be applied with the result field inside each returned tuple.

In this example, the client receives readings from the motes nearby and produces an ambient reading. A simple extension is to perform the same **rdg** operation but by specifying a different host, thus receiving the ambient information for another part of the system. In this way, the ambient values of different areas of the sensor field can be compared.

6 The TinyLime Middleware Architecture

The TinyLime model and programming interface have been designed as independent from the specific platform. However, our current implementation targets the Crossbow MICA2 mote platform, and exploits the functionality of TinyOS. On standard hosts, TinyLime is implemented as a layer on top of Lime without requiring any modification to it[4], reasserting the versatility of the Lime model and middleware. In this section we describe the internal architecture of TinyLime, whose main components are shown in Figure 7. It is worth noting that `MoteLimeTupleSpace`, `MoteLimeTuple`, `MoteLimeTemplate`, and the condition and aggregation classes 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 the motes. Our presentation begins just below the programming interface exported to the client and moves progressively toward the components deployed on the motes.

6.1 Interaction between Client and Base Stations

The first component we examine in detail is the `MoteLimeTupleSpace`, the primary user class. Although it presents to the client the illusion of a single tuple space containing sensor 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 different 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.

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

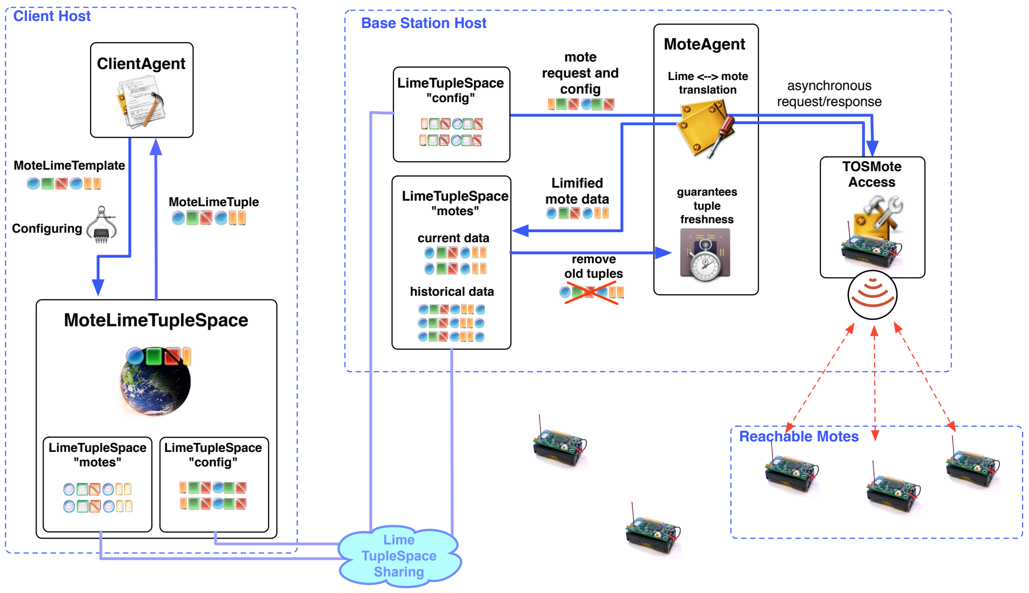

Fig. 7. Main architectural components on the base station and client hosts. Although shown separate here, the two can be co-located.

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 fresh match. If no such 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 tuple formats and reactions are defined internally to TinyLime. 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 removing stale tuples upon expiration of a timer. The *config* tuple space is also used for implementing the mote configuration requests described previously (e.g., `setActive`) using a similar scheme based on request tuples and reactions.

In TinyLime, all base stations run an instance of `MoteAgent`, a Lime agent that 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.

6.2 Interaction between Base Station and Motes

To this point we have described how data is retrieved by the client once it is available in the base station tuple space, 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 from the *config* tuple space, 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. Both read and reaction serve for both normal and aggregate values, but with additional parameters for the aggregates specifying the start/stop epochs and the aggregation function to be used. Requests which last an extended period of time, i.e. blocking reads and reactions, accept a listener parameter that 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. A reaction request can be canceled using the stop operation.

Communication. In principle, communication between the base station to and from motes is simple 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 requests. Because motes cannot receive packets while sleeping, the base station must repeatedly send a single packet as shown in Figure 8. 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. The *nominal awake time* is the amount of time that a mote promises to be awake during each epoch. The *epoch* is the basic cycle time of a mote described in Section 4. 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

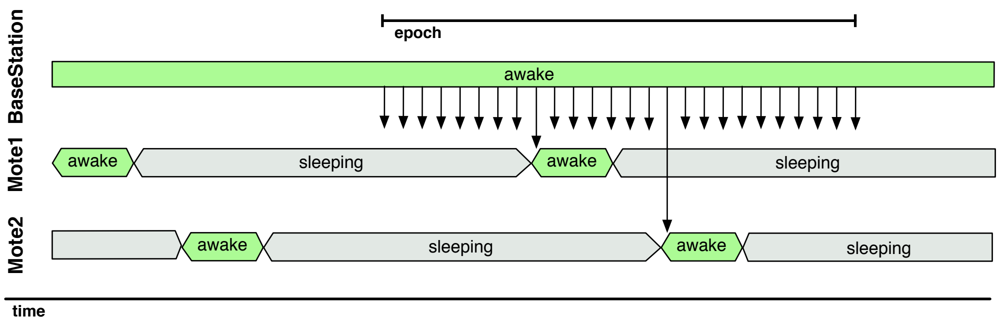

Fig. 8. Communication with motes that sleep for a majority of the time. Retransmission throughout an epoch ensures all motes in range receive each packet.

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 mote communication, we return to the `TOSMoteAccess` component. Sending a read or parameter set request to a mote is accomplished in this component by 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, the request is re-transmitted. 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**, are handled differently 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. As with regular communication, we chose 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 required on the motes to cancel the operation. When a client moves out of range of a 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.

6.3 On-Mote Components

The only remaining components are those deployed on the motes themselves. Given the choices made up to this point, the motes component is designed primarily as a reactive system, responding to incoming messages, managing its epoch and awake periods, and possibly logging sensor data for later aggregation. Figure 9 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 `Aggregation and Logging` component takes care of reading and recording sensor values that have been requested by clients for aggregation purposes. It also stores the aggregation functions available to the client. The `Sensors Subsystem` invokes the appropriate TinyOS components to take sensor readings, including powering sensors on and off before and after use. 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 setting parameters. The on-mote architecture is designed with an eye toward extensibility. Adding a new sensor requires changes only to the `Sensors Subsystem`. Aggregation functions can logging facilities can be modified separately or together.

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 from the packet. The `Core` then communicates with the `Aggregation and Logging` component. If a values has been recently logged, it is returned, otherwise a new reading is taken and an event is raised on the `Core` to pass the value up. The value is checked against the conditions contained in the packet and, if it 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.

If the request is for an aggregated value, the request is passed from the core to the `Aggregation and Logging` component where the requested aggregation function (e.g., average) is executed over the requested epoch range. The result is returned in the same manner as before.

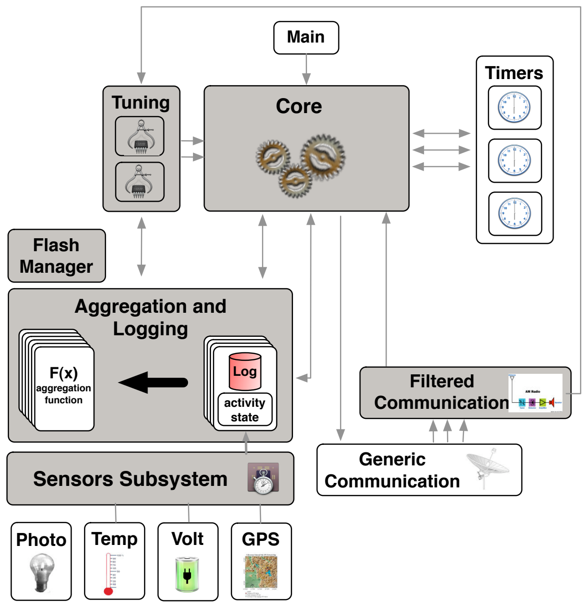

Fig. 9. Architecture of components installed on motes. Shaded components have been developed for TinyLIME, while the others are provided by TinyOS.

6.4 An Example — Reading a Single Sensor Value

To summarize how the components fit together in our middleware architecture, we walk through a simple example where a client reads a sensor light value, as shown in Figure 10. 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.

7 Implementation Details

TinyLIME is available for download at [tin, 2000]. Figure 11 shows the breakdown of source code lines across components.

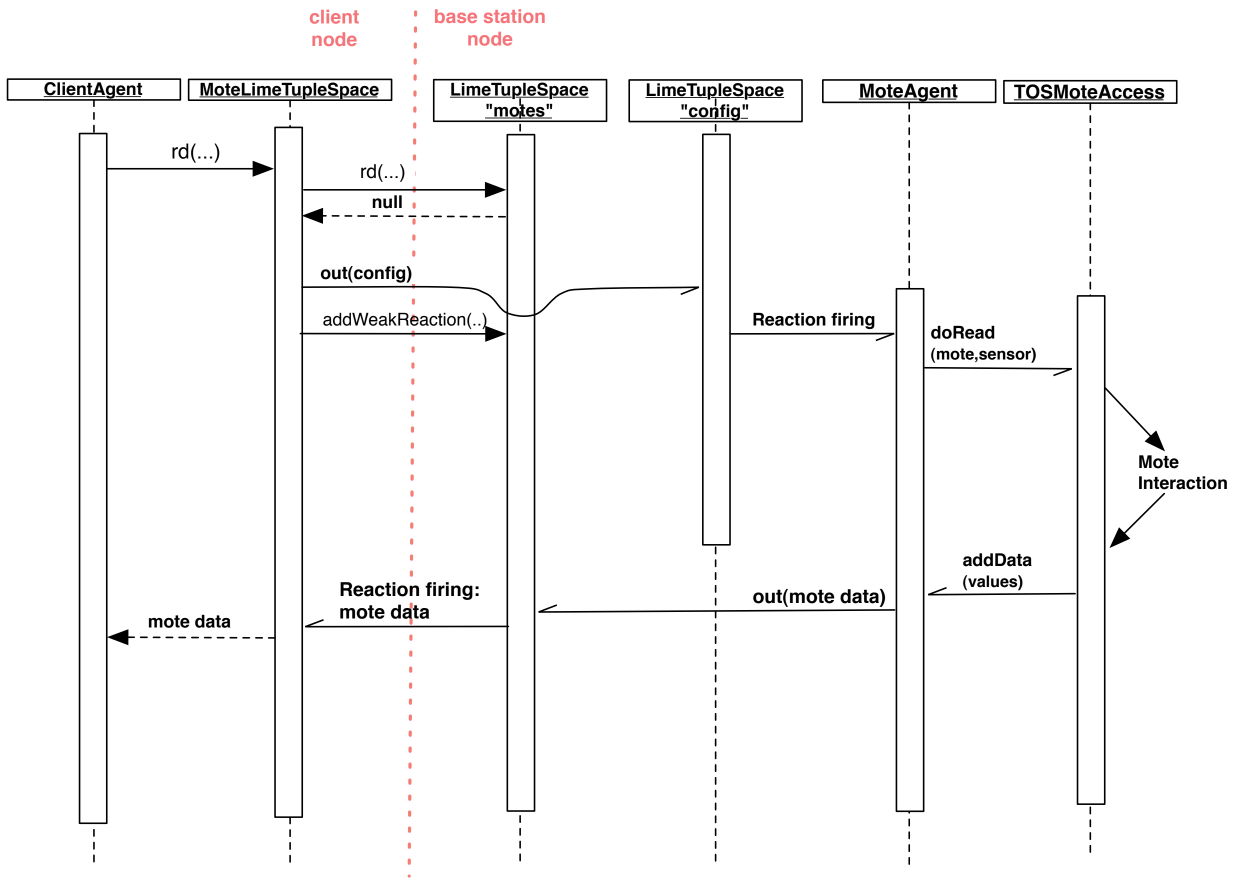

Fig. 10. Sequence diagram showing the processing of a **rdp**.

Component	Language	LOC
`MoteLimeTupleSpace`	Java	633
`MoteAgent`	Java	587
`TOSMoteAccess`	Java	520
On-Mote Components	nesC	614

Fig. 11. Uncommented lines of source code for all major TinyLIME components.

Aggregation and Logging. Although the concepts behind aggregation are simple, its implementation presents many degrees of freedom. First are the actual aggregation functions. We have implemented a suite of simple functions ranging from average to variance, however, the application programmer can easily expand this by inserting a new component into the `Aggregation and Logging` component on the motes. The return value of the function can also be modified to return any 64 bits. Our current functions use only 16 bits to return an integer, however, the use of the bits can be tuned by the application, for example creating a function that returns both the average, maximum, and minimum values for a specified epoch range. The result simply needs to be unpacked from the 64 bit integer returned in the third field of the tuple. Such bit packing and unpacking is not unusual when using limited architecture systems such as the motes.

Another set of choices comes in logging, both how much to log and where to store it. In our current implementation, the log is kept in RAM, using approximately 4KB (minus space for the runtime stack) for all data. Since each log entry requires 16 bits for the value and 32 bits for the epoch number, approximately 300 to 400 readings can be placed into the available space. One immediate source of expansion is to exploit the 512KB of flash data. Figure 9 shows the location of this management in the architecture, however we have not yet completed the integration.

The available memory can be divided among the sensor values in any manner appropriate to the application, for example giving more space to a sensor whose long term history is more important. We currently divide the space evenly, using each partition as a circular list for values from a single sensor.

Finally, when requesting an aggregated value, the user specifies both the minimum number of readings that can form an aggregate as well as the interval over which the aggregate should be performed. If the mote does not have sufficient data to aggregate over the full full requested range, the returned epoch range may be smaller than the request. For example, consider a request for an average over (100,0). A mote active only for the last 10 epochs cannot average 100 values, but only 10. The epoch interval returned will correspond only to the last 10 epochs. Similarly, if a mote recorded values from (100,90) and (10,0), the result will be the same because an average with a gap has little meaning. The precise interval returned and the semantics for addressing gaps are defined by the programmer for each aggregation function.

Evaluation. To get a feel for response times, we ran some experiments with an epoch time of $8s$ and an awake time of $2s$. These values, especially the epoch time, heavily influence the numbers below.

Our tests 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. Finally, our reported test results show queries for a regular sensor value, not an aggregated one. Because the computation time for aggregation is minimal with respect to communication, the results for aggregated and regular values are identical and therefore not shown. All tests were run with the TinyLime client co-located with the base station in order to eliminate 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 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 repeated the test with a second, immediate **rd** showing the same results as before. Again, this is expected 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 likely to be evenly distributed. This creates two problems. First, if a base station begins requesting data when all motes are sleeping, it may experience a long delay before some mote wakes up and replies. Second, if multiple motes receive the request and try to reply, despite the carrier sense and collision avoidance, multiple transmissions may saturate the channels and affect performance.

To avoid these situations, we have proposed a preliminary 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. An overview is available in [Curino et al., 2005].

Spreading Out Sensor Readings. As noted previously, when a mote is requested to start logging, readings are not taken every epoch, but instead at a sampling period specified at compile time. Consider a scenario with three motes close to one another, recording essentially the same data. Assuming they are all configured to log light values every three epochs, they may all log values in the first, fourth, seventh, etc. epoch from the moment the logging request arrived. However, it would be more useful to the application if the values with recorded spread out over time, for example with the first mote logging in the first, fourth, seventh, etc.; the second mote in the second, fifth, and eighth, and the third mote in the remaining third, sixth, and ninth epochs. This spreading can be achieved by exploiting the results of the wakeup scattering described earlier. Specifically, the earlier a mote wakes up in the epoch, the earlier in the sampling period the value is logged. In our example, this would imply that the first mote woke up earlier in the epoch than the second and the third even later. Because their wakeup times are evenly distributed, the sampling epochs are also evenly distributed.

8 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 [Abdelzaher et al., 2004], a middleware for environmental tracking applications, supports event-driven programming by identifying an event at a specific location in the sensed region, 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 across multiple sensors.

An alternative model is data-oriented and thus closer to TinyLime. In Directed Diffusion [Intanagonwiwat et al., 2000], 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 [Madden et al., 2003] provides an SQL-like interface with optimization for placement of parts of the query (e.g., joins, selects) to minimize power consumption. Cougar [Bonnet et al., 2000] and SINA [Shen et al., 2001] 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 [Li et al., 2003], 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 [Lazaridis et al., 2004] 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 [Bychkovskiy and Stathopoulos, 2002] 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 [Jonsson, 2003] exploits multiple tuple spaces, each holding only locally sensed 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.

9 Conclusions

In this paper we described a new middleware, TinyLime, supporting the development of sensor network applications. TinyLime is an extension of the Lime middleware, originally designed for MANETs. The adaptation of Lime to the sensor network environment necessitated not only changes in the implementation to handle the restricted platform of the sensors, but also changes in the model to introduce sensors as new components in the system. The result, TinyLime, is implemented as a layer on top of Lime with specialized components deployed on sensors and base stations. It supports application-tunable energy utilization by allowing the user to turn on and off logging of values on the sensors. Logged values are available for efficient aggregation on a single sensor while the basic TinyLime operations support aggregation across multiple 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

LIME Web page. `http://lime.sourceforge.net`, 2000.

TinyLIME Web page. `http://lime.sourceforge.net/tinyLime.html`, 2000.

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

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.

G. Anastasi, M. Conti, E. Gregori, A Falchi, and A. Passarella. Performance Measurements of Mote Sensor Networks. In *Proc. of the ACM/IEEE Int. Symp. on Modeling, Analysis and Simulation of Wireless and Mobile System (MSWiM'04)*, October 2004.

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

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.

C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, and G.P. Picco. TINYLIME: Bridging Mobile and Sensor Networks through Middleware. In *Proc. of the 3rd IEEE Int. Conf. on Pervasive Computing and Communications (PerCom 2005)*, pages 61–72, Kauai Island (Hawaii, USA), March 2005. IEEE Computer Society.

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

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.

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.

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

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

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.

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.

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.

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.

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.

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.

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

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

B. Sundararaman, U. Buy, and A.D. Kshemkalyani. Clock Synchronization in Wireless Sensor Networks: A Survey. *Ad-Hoc Networks*, 3(3):281–323, May 2005.