Using Coordination Middleware for Location-Aware Computing: A LIME Case Study

Amy L. Murphy[1,2] and Gian Pietro Picco[2]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
E-mail: picco@elet.polimi.it
[2] Department of Computer Science, University of Rochester, NY, USA
E-mail: murphy@cs.rochester.edu

Abstract. The decoupling between behavior and communication fostered by coordination becomes of paramount importance in mobile computing. In this scenario, however, coordination technology is typically used to deal only with the application data necessary to orchestrate the process activities. In this paper, we argue instead that the very same coordination abstractions can be used effectively to deal also with information coming from the physical *context*—a fundamental aspect of mobile computing applications.
We cast our considerations in LIME, a coordination model and middleware designed for mobile computing. To support our arguments with concrete examples, we report about the development of TULING, a proof-of-concept application enabling the tracking of mobile users. The lessons learned during development enable us to assess the feasibility of the approach and identify new research opportunities.

1 Introduction

Mobile computing defines a challenging environment for software development. Communication is enabled by wireless links, which are less reliable and intrinsically dependent on the relative positions of the mobile parties in communication. Similarly, location affects the overall *context* perceived by a mobile unit, by constraining not only the available communication parties, but also the data available for computation, the set of accessible services, and in general the resources available to a component.

It has been observed [13] that software development in the mobile environment can be tackled successfully by exploiting a coordination perspective. The decoupling between behavior and communication fostered by coordination enables one to separate the applicative behavior of components from the continuously changing context in which they are immersed. Examples of systems that have applied this intuition to deal with physical mobility of hosts or logical mobility of agents include Klaim [10], XMIDDLE [7], Tucson [12], Mars [2], and LIME [8]. In all of these systems, a shared data structure—typically a tuple space—is used to store the data available to mobile units and to represent naturally the context available to them. Nevertheless, all of these systems focus

on providing support for coordination through a context consisting essentially of application data. Little or no support is provided for constraining the application behavior based also on the physical context.

Clearly, this is a limitation. Dealing with a changing physical context is fundamental in many mobile applications. Physical context information can be very diverse, and include local system information such as battery level or signal-noise ratio, or environmental information such as light intensity, temperature, or ambient noise. Among all, *location* is possibly the most relevant context element, in that it often *qualifies* the values of the others. For example, a temperature reading becomes more meaningful when accompanied by the identity of the room where it was sensed. The point, however, is that the actions of an application component in a mobile environment may depend on one or more of these context information values and modeling physical context becomes a necessity.

At the opposite extreme of coordination approaches, several middleware systems have been proposed that explicitly tackle the problem of managing a dynamically changing context. Relevant examples include the Context Toolkit [3], Odyssey [11], Aura [5], Gaia [14], and Owl [4]. The focus of these systems is on allowing applications to retrieve information about context, either proactively (by directly querying some context representation) or reactively (by subscribing to changes in the context information). The middleware takes care of properly disseminating the context information to the involved parties, and hence greatly simplifies the management of physical context used in mobile applications. On the other hand, these systems provide little or no support for representing and managing the data context, used for coordinating the behavior of the application components.

In this paper, we argue that the gap between the two aforementioned perspectives can be reduced, if not eliminated, by exploiting coordination abstractions also for the management of the physical context. Once acquired by appropriate sensors, context information is essentially like any other data, and hence can be treated as such in a data-centric coordination approach. The very same primitives used to manipulate, retrieve, or react to available data for the sake of coordination can now be used for dealing with physical context information. These two traditionally separate dimensions are unified under a common set of abstractions, simplifying considerably the design and implementation of mobile applications.

Although our considerations are in principle applicable to any data-centric coordination approach, in this paper they are cast in LIME [8], a coordination model and middleware expressly designed for mobile computing. A concise overview of LIME is provided in Section 2. Section 3 illustrates the premise of our approach, and describes how a coordination middleware can be used to disseminate physical context information. The implications of our design approach can be understood and assessed only through the reality check provided by application development. Hence, to support our arguments with concrete examples, in Section 4 we report about the design and implementation of TUPLING, a proof-of-concept application providing location tracking of mobile users. The lessons learned from this ex-

perience enable us to assess, in Section 5, the feasibility of the approach and identify new research opportunities. Finally, Section 6 ends the paper with brief concluding remarks.

2 LIME: Linda in a Mobile Environment

The LIME model [8] defines a coordination layer for applications that exhibit logical and/or physical mobility, and has been embodied in a middleware available as open source at `http://lime.sourceforge.net`. LIME borrows and adapts the communication model made popular by Linda [6].

In Linda, processes communicate through a shared *tuple space*, a multiset of tuples accessed concurrently by several processes. Each tuple is a sequence of typed parameters, such as `<"foo",9,27.5>`, and contains the actual information being communicated. Tuples are added to a tuple space by performing an **out**(t) operation. Tuples are anonymous, thus their removal by **in**(p), or read by **rd**(p), takes place through pattern matching on the tuple content. The argument p is often called a *template*, and its fields contain either *actuals* or *formals*. Actuals are values; the parameters of the previous tuple are all actuals, while the last two parameters 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, selection is non-deterministic.

Linda characteristics resonate with the mobile setting. Communication is implicit, and decoupled in *time* and *space*. This decoupling is of paramount importance in a mobile setting, where the parties involved in communication change dynamically due to migration, and hence the global context for operations is continuously redefined. LIME accomplishes the shift from a fixed context to a dynamically changing one by breaking up the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of the individual tuple spaces based on connectivity.

Transiently Shared Tuple Spaces. In LIME, a mobile unit accesses the global data context only through a so-called *interface tuple space* (ITS), permanently and exclusively attached to the unit itself. The ITS, accessed using Linda primitives, contains tuples that are physically co-located with the unit and defines the only data available to a lone unit. Nevertheless, this tuple space is also *transiently shared* with the ITSs belonging to the mobile units currently accessible. Upon arrival of a new unit, the tuples in its ITS are merged with those already shared, belonging to the other mobile units, and the result is made accessible through the ITS of each of the units. This sequence of operations, called *engagement*, is performed as a single atomic operation. Similarly, the departure of a mobile unit results in the *disengagement* of the corresponding tuple space, whose tuples are no longer available through the ITS of the other units.

Transient sharing of the ITS is a very powerful abstraction, providing a mobile unit with the illusion of a local tuple space containing tuples coming from all the

units currently accessible, without any need to know them explicitly. Moreover, the content perceived through this tuple space changes dynamically according to changes in the system configuration.

The LIME notion of a transiently shared tuple space is applicable to a mobile unit regardless of its nature, as long as a notion of connectivity ruling engagement and disengagement is properly defined. Figure 1 shows how transient sharing may take place among mobile agents co-located on a given host, and among hosts in communication range. Mobile agents are the only active components, and the ones carrying a "concrete" tuple space; mobile hosts are just roaming containers providing connectivity and execution support for agents.

Operations on the transiently shared tuple space of LIME include those already mentioned for Linda, namely **out**, **rd**, and **in**, as well as the probing operations **rdp** and **inp** whose semantics is to return a matching tuple or return `null` if no matching tuple exists at the time the query is issued. For convenience LIME also provides the bulk operations **rdg** and **ing** that return a set of tuples that match the given pattern. If no matching tuples exist, the set is empty.

Restricting Operation Scope. The idea of a transiently shared tuple space reduces the details of distribution and mobility to changes in what is perceived as a local tuple space. This view is powerful as it relieves the designer from specifically addressing configuration changes, but sometimes applications may need to address explicitly the distributed nature of data for performance or optimization reasons. For this reason, LIME extends Linda operations with scoping parameters, expressed in terms of agent or host identifiers, that restrict operations to a given projection of the transiently shared tuple space.

The $\mathbf{out}[\lambda](t)$ operation extends **out** by allowing the programmer to specify that the tuple t must be placed within the tuple space of agent λ. This way, the default policy of keeping the tuple in the caller's context until it is withdrawn can be overridden, and more elaborate schemes for transient communication can be developed. Location parameters are also used to annotate the other operations to allow access to a slice of the current context. For instance, $\mathbf{rd}[\omega, \lambda](p)$ looks for tuples matching p that are currently located at ω but destined to λ. LIME allows ω to be either a host or an agent, enabling either an entire host-level

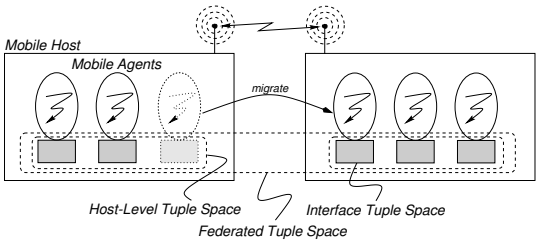

Fig. 1. Transiently shared tuple spaces encompass physical and logical mobility.

tuple space to be queried or a subset of the host-level tuple space containing the tuples specific to the named agent.

Reacting to Changes. In the dynamic environment defined by mobility, reacting to changes constitutes a large fraction of application design. Therefore, LIME extends the basic Linda tuple space with a notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s specifying the actions to be performed when a tuple matching the pattern p is found in the tuple space. A notion of *mode* also is provided to control the extent to which a reaction is allowed to execute. A reaction registered with mode ONCE is allowed to fire only one time, i.e., after its execution it becomes automatically deregistered. Instead, a reaction registered with mode ONCEPERTUPLE is allowed to fire an arbitrary number of times, but never twice for the same tuple. Details about the semantics of reactions can be found in [8] and a formal semantics is available in [9]. Here, it is sufficient to note that two kinds of reactions are provided. *Strong reactions* couple in a single atomic step the detection of a tuple matching p and the execution of s. Instead, *weak reactions* decouple the two by allowing execution to take place eventually after detection. Strong reactions are useful to react locally to a host, while weak reactions are suitable for use across hosts, and hence on the entire transiently shared tuple space.

Exposing System Configuration. With the perspective presented thus far, applications are aware only of changes in the portion of context concerned with application data. Although this is enough for many mobile applications, in others knowing which hosts are connected also plays a key role. For instance, a typical problem is to react to departure of a mobile unit, or to determine the set of units currently belonging to a LIME community. LIME provides this form of awareness of the system configuration by using the same abstractions already discussed: through a transiently shared tuple space conventionally named LimeSystem to which all agents are permanently bound. The tuples in this tuple space contain information about the mobile units present and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, which host they reside on. Insertion and withdrawal of tuples in the LimeSystem is the prerogative of the runtime support. Nevertheless, applications can read tuples and register reactions to respond to changes in the configuration of the system.

3 Representing Physical Context in LIME

Applications written for the mobile environment must have access to their own data as well as the data of the other components within range. Providing this access while ignoring the details of communication is the primary goal of LIME and of other coordination systems targeting the mobile environment. In addition to application data, many mobile applications are characterized by their need to respond to the environment in which they find themselves, adapting application behavior as their location, bandwidth, or environmental parameters change. In other words, they must be able to react to changes in the physical context.

The LimeSystem tuple space described previously shows that the transiently shared tuple space style of coordination promoted by LIME can be used successfully to provide mobile applications access to one form of context, namely the identity of accessible hosts and agents. Our goal here is to show that LIME can similarly be used to represent, share, and interact with other aspects of context as well. We further show that by placing context information into a transiently shared tuple space applications obtain easy access to what we refer to as the *distributed context*, or the context of all components within range.

At its core, our idea is simple: insert context information into a transiently shared tuple space thus allowing all connected components to access it through the proactive and reactive constructs of LIME. In this section we describe the representation of context information in the tuple space, the primary access mechanisms, and the benefits provided to the application programmer.

Making Context Accessible. Our work focuses on the sharing of and interaction with context data through tuple spaces, intentionally leaving aside the issues related to detecting context information as well as the format of that information. Therefore, we do not focus on any specific sensing technologies nor even on any specific type of context information. Instead, we provide a general infrastructure to exploit and *disseminate* context, and that can be easily adapted to the needs of any system. The key component of our infrastructure providing this isolation is a context agent that interacts with the sensors and with LIME.

The context information to be made available is represented as tuples, identical in all respects to traditional application data tuples. Interestingly, this can be exploited to represent single values as well as sequences of temporally related values to maintain history or for aggregation of context information. In principle, it is therefore possible to mix data and context tuples in a single tuple space. This, however, leads to a cumbersome design that imposes restrictions on applications (such as requiring that no application tuple use the same pattern as any context tuple) and mixes the interaction with the two types of data. Therefore, ideally the two kinds of data should be insulated from one another. LIME supports this separation with a mechanism for creating multiple tuple spaces, uniquely identified by name, and whose contents are independent. By creating a separate *context tuple space* (e.g., named CONTEXT), application tuple formats remain unrestricted and interaction with context information is made explicit by issuing operations on the context tuple space.

For applications residing on the same host as the context agent, the context information is accessible locally by issuing the normal LIME operations on a CONTEXT tuple space. Such interaction enables an application to query for the current context, and attach this to generated data. For example, a video player can adjust its buffer size based on the currently available local memory. This memory level information is retrieved from the local context tuple space.

Because LIME tuple spaces are transiently shared, the information in the context tuple space is also available to all connected components, yielding what we refer to as the *distributed context*. For example, firefighters can spread location-enabled temperature sensors throughout a forest fire and annotate a map with

the temperature gradient from the information available in the distributed context. Even if only a limited number of sensors are within range, those connected provide the context for the area immediately surrounding the firefighter generating the map.

Interacting with Context. Middleware, and in general systems targeted toward the mobile environment, have evidenced that access to context information should be both proactive and reactive, meaning that a program should be able to *pull* the information on demand or have it *pushed* whenever it changes. By representing the context information inside a LIME tuple space, both styles of operation are available. Proactive operations map to the query operations of LIME (e.g., **rdp**) while reactive mechanisms are enabled by strong and weak LIME reactive statements. Furthermore, the LIME extensions to control scope enable operations over the entire distributed context or over a projection of the context tuple space, tailoring operations to apply to all hosts in range or only a single host.

For applications such as the video player described previously, the query operation **rdp**, restricted in scope to the application's own host-level tuple space, returns the needed current available memory. The same operation can be used to retrieve the context of a remote host simply by changing the scope parameters. LIME bulk operations, such as **rdg**, are especially useful to retrieve historical context. For example, an application validating the functioning of the air conditioning in a building can query for the history of temperatures in a given room.

In our previous LIME programming efforts, we found that although the query operations are useful, the core application functionality often utilizes reactive constructs. With respect to context information, reactions are most useful for monitoring and immediately adapting to changes. For example, a reaction over the host's own context tuple space can notify an application when its battery power is low. Changing the scope of the reaction to monitor a remote host's battery can trigger a modification in the mode of interaction between the two hosts. Alternately, to keep the map of temperatures up to date, a reaction over the distributed context can be registered to fire every time any sensor's temperature value changes.

Finally, the distributed context can easily be searched, providing a dimension of accessibility not present in most context systems. For example, a query can be formed to find one or all components at a specific location in space. This powerful operation uses the same LIME primitives as before, **rdp** in this case, and does not require any kind of server support, a solution common with other context-aware systems. Such serverless operation makes our solution for managing context more amenable for mobile ad hoc networks, and other highly dynamic scenarios.

Benefits to the Application. The ideas presented here unveil how a variety of context information can be represented in the context tuple space and accessed by connected components using the usual LIME primitives. This approach has two main benefits. First, the management of context is decoupled from the applications that use it. This implies that the context maintenance mechanisms

can easily be substituted or extended to present more context information from additional sensors. For example, if an application is designed to use location information to build a map of connected users, the location context information can be changed from GPS to an indoor infrared tracking system without changes to the application as long as the tuple format does not change.

Second, placing context information into a tuple space unifies the management of application data exploited for coordination, and of the physical context perceived by the application. This allows programmers to deal with both using the same interface with evident benefits in terms of ease of development and understanding of the resulting implementation.

4 TULING: Tracking Users in LIME with GPS

To further explore and validate our ideas about representing context in LIME, we chose to focus on a single aspect of context, namely location in physical space. This choice is motivated by the observation that location is critical to many mobile applications. Often, context information is dependent on the particular location where it is sensed. Moreover, location has a value *per se*, in that it provides a direct and intuitive way to express mobility of users.

In this section, we discuss the design and implementation of TULING, an application for collaborative exploration of a space. Although we have already discussed the main idea for representing and interacting with context in LIME, it is only through the elaboration of these ideas in a real application that they can be fully appreciated and we can report about the ease of incorporating location context into an application.

TULING is intended to be used by multiple individuals moving through a common environment, each equipped with a GPS- and wireless-enabled PDA. While the immediate goal of this application is to provide a proof-of-concept for our approach, we can envision the functionality provided as useful in several real-world scenarios, e.g., coordination of a rescue team deployed in a in a disaster scenario. The major portion of the display of each user, as seen on the left of Figure 2, is a representation of the user's current location in space, where a sequence of dots indicates her past movement itinerary. When a new user comes within range, her name is displayed in the box on the top right of the display. To allow users to coordinate their actions as they move, each user can specify a monitoring mode for viewing the movement of the others by first selecting the user, then pressing one of the buttons to the top right.

By pressing the *monitor* button, a user's movements are marked on the screen as long as she is connected. When the user disconnects, her name disappears from the list, but the dots remain on the screen. In monitor mode, the user's displayed location is kept as up to date as possible, but only the itinerary during connection is tracked. To retrieve the history of movement that occurred before connection, a user must press the *getItinerary* button. Instead of being a continuous operation such as monitoring, this is a one-time operation that displays the entire history of movement (up to that moment) for the selected user. Another one-time operation

is to display the current location of a user by pressing the *getCurrent* button. The final mode is simply to ignore a user.

In addition to these monitoring modes, TULING provides two other that apply globally to all users that come within range. These modes are available as menu options and are designed to always retrieve the history for every users, and to actively monitor the movement of every user within range. When both these modes are active, a user's display shows the entire itinerary of every user she has encountered, i.e., including movements that occurred before and during connection. In a disaster recovery scenario, this mode can be useful for a supervisor to monitor the movements of all the members of a team around her.

TULING also allows users to add annotations, such as a textual note or a digital photograph, to their own current location. For instance, after an earthquake,

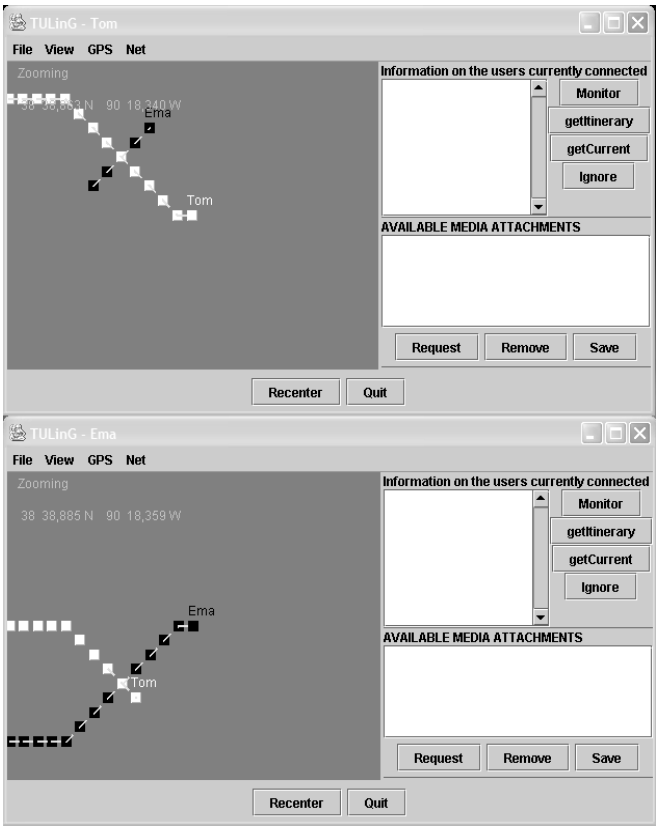

Fig. 2. Screenshot of two TULING users, Tom (the top image) and Ema, after disconnection. Before the two users were disconnected, each was monitoring the location of the other, but only Ema had retrieved the history of Tom. After disconnection, the movements of the users are not visible to one another, but only to themselves.

annotations could take the form of photographs of damaged walls or descriptions of work to be completed at a location. These annotations are indicated on the display with a special icon: by clicking on the icon, the annotation can be viewed as long as the user is connected.[1]

As already noted, TULING can be useful in disaster recovery scenarios for allowing workers to coordinate their actions. For example, workers monitoring the current movements of others in the area can either avoid one another in order to explore different territory, or gather to discuss face to face their findings. Annotations created by survey workers can be evaluated by aid workers to direct supplies to specific areas. The benefits of using TULING include the ability for users to interact without requiring line-of-sight, the enabling of impromptu sharing of information without the required support of a central server, and the variety of monitoring modes to adjust the communication overhead, and thus battery requirements, on a per-user basis.

As a proof-of-concept application, TULING demonstrates how the operations available in LIME are both natural and sufficient to provide the range of interaction necessary for exploiting context, as we illustrate in the remainder of this section.

Representing and Updating Location Context. The combined requirements to both monitor the current location of a user and to display the previous itinerary require that TULING provide access to the current location and to the previous locations of a user. The *current location* is represented by a single tuple containing the GPS coordinates and a timestamp. To update this tuple, we first insert the new location tuple, then remove the old. The motivation for this sequence of operations is to ensure that a location is always accessible to a probe operation. It is true that the old value may be returned instead of the new, but the two values are unlikely to be significantly different, and furthermore the timestamp can be examined for freshness.

Because a host has only a single current location, we needed a separate tuple representation for the location *history*, i.e., the user itinerary. We explored the idea of having one tuple for each previous location, simply changing the pattern to include a new field with the label HISTORY. This solution turns out to be unreasonable because the overhead necessary to retrieve the entire history of a user is proportional to the number of tuples retrieved. Therefore, we opted for grouping multiple locations together into a single *stride* tuple that contains a sequence number and a list of locations. The number of locations in the stride list is tunable. We chose a value that balances the overhead of retrieving all the stride tuples to build a history with the overhead of updating the stride tuple with each new location. In other words, to keep the entire history in the tuple space, each time a new location is generated the most recent stride tuple, identified by a sequence number, is removed, updated, and reinserted into the

[1] The choice of making annotations inaccessible while users are disconnected was motivated by the performance considerations discussed later. If needed by an application, this choice and be easily reversed, but with an accompanied increase in overhead.

tuple space. This solution also opens up opportunities for implementing "garbage collection" of *old* stride tuples.

It is worth noting that both the updating of the current and stride tuples are operations entirely local to the agent performing them. Therefore they are executed with very low cost and generate low system overhead.

Accessing Location Context. TULING uses a combination of reactions and probe operations, both over the federated tuple space as well as specifically scoped, to implement both the functionality of the buttons on the top right of the display and the alternate modes to monitor and retrieve the histories of all users.

To our surprise, the LimeSystem continued to be an integral part of the operation of TULING. For example, to display the name of each user within range, TULING employs a reaction on the LimeSystem tuple space. By exploiting the fact that LIME keeps this list up to date, users are able to select monitoring modes on a per user basis, as well as see, at a glance, which users are connected.

As already mentioned, the *monitor* mode allows a user to be tracked while they are connected. This essentially involves reacting to a change in the user location and updating the display—an operation that naturally calls for a LIME reaction. Therefore, the tracking functionality is implemented with the installation of a ONCEPERTUPLE weak reaction restricted to the projection of the tuple space of the selected user. The pattern of the reaction is that of current location tuples.

The functionality of both the *getItinerary* and *getCurrent* buttons are implemented with probe operations over the projection of the tuple space of the selected user. The first exploits a **rdg** operation for stride tuples to retrieve *all* of the tuples of the itinerary. The latter uses a similarly scoped **rdp** for retrieving the current location tuple.

These explicitly selected modes offer the ability to tailor monitoring on a per-user basis, constructing her view of the environment and controlling the amount of system resources dedicated to monitoring each user. If such tight controls are not necessary, the two options for monitoring all users and automatically downloading the histories of all users can be enabled. Monitoring all users in LIME is accomplished with a single ONCEPERTUPLE weak reaction registered on the transiently shared tuple space, with the pattern of the current location tuples. The reaction fires each time any user's location changes, updating the display accordingly.

As in the case of a specific user's itinerary, retrieving all itinerary information also uses the bulk operation **rdg** to query a specific user's projection of the tuple space for all stride tuples. To accomplish this for *all* users, TULING registers a reaction on the LimeSystem tuple space to fire each time any host arrives. This, in turn, causes the **rdg** to be invoked as part of the reaction code.

Exploiting Location. While the previously described functionality is designed to show how context information can be shared and visualized, the annotation feature of TULING demonstrates how applications can attach location context to application data.

Annotations can be either simple text, or a file containing, for example, an image. After the annotation has been created, TULING associates it with the user's most recent location. To find it, TULING queries the local context tuple space using a probing read, **rdp**. This returns a tuple containing the timestamp and the current location coordinate. These pieces of information, along with the annotation are combined into a single annotation tuple, which is inserted into a regular LIME tuple space, making it accessible to other users.

One of the requirements of TULING is to display the existence of an annotation as an icon on top of the normal location icon. With the implementation of annotations just described, the only way to do this is to query for the actual annotation tuples at the same time that the itinerary tuples are retrieved, or to react to annotations when a host is being monitored. If the annotations themselves are large, this creates an unreasonable amount of overhead, especially wasteful if the annotations are not viewed eventually by the user. Therefore, we modified the representation of annotations in the tuple space, effectively creating two tuples with the same information as the original tuple. The first tuple, contains only the location and an annotation identifier. It is this tuple that is retrieved and monitored in the cases above, reducing the overhead because the identifier is small in comparison to a typical annotation. The second tuple contains the annotation identifier and the actual annotation. This tuple is retrieved on demand when a user opts to view the annotation. The result, however, is the restriction that annotations can only be viewed while users are connected, a reasonable compromise for effectively managing overhead.

5 Discussion and Lessons Learned

Experience with TULING provides evidence to support our argument for placing context information into the tuple space. In this section we discuss the relationship between our work and others and discuss extensions to LIME that would better support context.

Related Work. The unification of access to both data and context is a factor that distinguishes our work from that of other systems for accessing context. For example, the Context Toolkit [3] wraps context providers with a standard interface that provides query and notification access. In LIME, these two modes of interaction are naturally provided by the query and reaction operations, the same operations used for application data access.

Furthermore, the Context Toolkit provides a separate service discovery server that identifies context providers to which the programmer can explicitly bind to receive context information. By using LIME, this type of centralized service discovery is not necessary. Instead, the currently available components are identified from the fully distributed LimeSystem tuple space, which is also accessed by the same tuple space operations. Finally, in LIME, while it is possible to access specific context providers by limiting the scope of the operations, it is also possible to access the entire distributed context with the LIME primitives, something

that while possible in some other systems, would require additional, non-trivial programming effort.

Placing context information in a LIME tuple space also simplifies access and reaction to the context of remote components. This is fundamentally different from the context reaction mechanisms provided by Odyssey [11]. In Odyssey, a component can register to be notified when the local context changes, allowing, for example, a server to detect a bandwidth reduction. However, Odyssey is not designed for servers to monitor context changes at the mobile client. In LIME, instead, registering for remote changes is as straightforward as registering for local changes. Furthermore, the implementation of weak reactions ensures reasonable performance.

The work presented here is also related to the many other coordination approaches designed to address the concerns of the mobile environment. We chose LIME as our foundation both because we are most familiar with it, and because it is well suited to the needs of interacting with context. In principle, it should be possible to apply our idea of representing context as data in other coordination systems, although we are not aware of any work from the coordination community in this direction.

LIME *Extensions.* While the current implementation of TULING well serves as a proof-of-concept, the development process illuminated several directions for extending LIME to better support interaction with context information, and most likely with application data as well. These include the ability to atomically change the contents of a tuple, to sequentially order a set of tuples, to replicate data, and to search for tuples based on a range. These features were initially left out of LIME, in an effort to strive for a minimal and yet expressive application programming interface. In the light of their relevance for improving handling of the physical context, however, we are currently reconsidering this decision and planning to include these features in the next release of our middleware.

As explained in Section 4, the updating of the current location field is accomplished by first outputting the new tuple, then removing the old tuple. The result is that in the short interval between the two operations, the component has *two* current locations, and a probe operation such as **rdp** is just as likely to return the old tuple as it is the actual current tuple. Reversing the operations, however, leaves an interval where a component has no location and **rdp** may return `null`. Both of these conditions could be avoided if LIME provided either an atomic "change" operation, performing **in** and **out** in the same atomic step, or a generic transaction construct for grouping arbitrary LIME operations.

When TULING retrieves itinerary tuples to display the history of movement of a user, the LIME bulk operation **rdg** returns the set of all stride tuples. Although this set does not have an inherent ordering according to LIME and Linda semantics, in TULING an order is implied by the stride sequence number. Currently, the application uses this value to sort the stride tuples before visualizing them. It would be convenient if LIME bulk operations were extended to allow the specification of a field over which an ordering is automatically imposed, thus releasing the programmer from the burden of coding ordering explicitly.

Another feature to consider adding to LIME is replication. In TULING, the previous locations of the other components are effectively replicated at the application level, to enable its visualization. Location information, however, is not duplicated within the tuple space. Therefore, if A copies B's history, and then later meets C, the information about B is outside the tuple space and therefore not accessible to C. Several efforts in the mobile ad hoc community have looked at the issue of replication [1, 7, 15], but none of the solutions is immediately applicable to the tuple space environment.

Finally, a useful operation to add to TULING is to support a query such as "find all components within a radius r from point (x, y)." Such a query requires a range search inside the tuple space, while LIME provides only value matching. Supporting this functionality requires a change in the tuple space implementation underlying LIME, as well as a change in the interface to the tuple space. Nevertheless, this would provide a great improvement in the expressiveness of the system. In TULING, the user could monitor an area around herself, or even request information in an area away from her current position.

6 Conclusions and Future Work

Our evaluation thus far has focused on location context. While location is useful for many applications, other aspects of context also have direct use for mobile applications. For example, by providing available bandwidth, applications can adjust their use of the tuple space, querying for large tuples only when the bandwidth is plentiful. By exposing context such as available storage space, applications can control how much data is moved to a particular tuple space. It is our belief that most of the effort to provide this extra context information is in the collection of the data itself, not in the presentation of the data context in LIME. Our work with location provides a model for this transformation as well as solutions for issues such as large histories.

In this paper, we have presented the requirements for presenting and accessing context information with coordination mechanisms and a description of TULING, an application that demonstrates how location context can made available in LIME. Throughout, our goal was to provide evidence to support the use of coordination middleware for enabling both local and remote components to share and interact with one another's context information. This work fills a gap between coordination systems that focus on providing access to application data and context-aware toolkits that concentrate only on enabling interactions with context. Furthermore, it effectively demonstrates that context information and application data can be treated in a unified manner and accessed with the same coordination operations. The result is a significant reduction in the programming effort to develop mobile applications that require access to physical context.

Acknowledgements. The work described in this paper was partially supported by the projects VICOM and IS-MANET, funded by the Italian government. The

authors wish to thank Emanuele Cordone and Thomas Pengo for their work on the implementation of TUPLING.

References

1. M. Boulkenafed and V. Issarny. A middleware service for mobile ad hoc data sharing, enhancing data availability. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro (Brazil), June 2003.
2. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. of the 2^{nd} Int. Workshop on Mobile Agents*, LNCS 1477. Springer, 1998.
3. A.K. Dey, D. Salber, and G.D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal, special issue on Context-Aware Computing*, 16(2-4):97–166, 2001.
4. M.R. Ebling, G.D.H. Hunt, and H. Lei. Issues for context services for pervasive computing. In *Proceedings of the Workshop on Middleware for Mobile Computing*. IFIP/ACM, 2001.
5. D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, April-June 2002.
6. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
7. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Kluwer Personal and Wireless Communications Journal*, 21(1), April 2002.
8. 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 21^{st} Int. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.
9. A.L. Murphy, G.P. Picco, and G.-C. Romjan. LIME: A coordination middleware supporting mobility of hosts and agents. Technical Report WUCSE-03-21, Washington University, Department of Computer Science, St. Louis, MO (USA), 2003.
10. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.
11. B.D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, 4, 1999.
12. A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *Proc. of the 1999 ACM Symp. on Applied Computing (SAC'00)*, February 1999.
13. G.-C. Roman, A.L. Murphy, and G.P. Picco. Coordination and Mobility. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 254–273. Springer, 2000.
14. M. Romn, C. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):65–67, 2002.
15. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Operating Systems Review*, 29(5):172–183, 1995.