Developing Mobile Applications: A LIME Primer

Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman*

August 12, 2003

Abstract

Mobility poses peculiar challenges that must be addressed by novel programming constructs. LIME (Linda in a Mobile Environment) tackles the problem by adopting a coordination perspective inspired by work on the Linda model. The context for computation, represented in Linda by a single, globally accessible, persistent tuple space, is reinterpreted in LIME as the transient sharing of the tuple spaces carried by individual mobile units. Additional constructs provide increased expressiveness, by enabling programs to deal with the location of tuples and to react to specified states. The resulting model provides a minimalist set of abstractions that promise to facilitate rapid and dependable development of mobile applications. In this paper, we illustrate the model underlying LIME, present the programming interface of the companion middleware, and discuss how applications and higher-level middleware services can be built using it.

1 Introduction

Distributed computing has been traditionally associated with a rather static environment, where the topology of the system is largely stable, and so is the configuration of the deployed application components. Today, this vision is being challenged by various forms of mobility, which are effectively reshaping the landscape of modern distributed computing. On one hand, the emergence of wireless communication and portable computing devices is fostering scenarios where the physical topology of the system is continuously modified by the free movement of the mobile hosts, whose wireless communication devices enable them to dynamically create and sever links based on proximity. In its most radical incarnation, represented by *mobile ad hoc networks* (MANET), the system is entirely constituted by mobile nodes, and the fixed network infrastructure is totally absent. Together with this *physical mobility* of hosts, another form of mobility has emerged, where the units of mobility are program fragments belonging to a distributed application, and are relocated from one host to another. This *logical mobility* of code brings unprecedented levels of flexibility in the deployment of application components, and in some domains it enables significant improvements in the use of communication resources.

Mobility undermines several common assumptions. Disconnection is no longer an infrequent accident: in applications involving physical mobility it is often triggered by the user in order to save battery power, and hence becomes a defining characteristic of the environment. The fluidity of the physical and logical configuration of the system renders it impractical—and often impossible—to make assumptions about the availability of a specific user, host, or service. In general, the computational context is no longer fixed and predetermined, rather it becomes continuously changing in largely unpredictable ways. As a consequence, a large fraction of the body of theories, algorithms, and technology must be recast in the mobile scenario. Application development requires appropriate constructs and mechanisms to accommodate the required level of dynamicity and decoupling required to cope with mobility. Nevertheless, thus far the problem has been tackled only in a limited context. Pioneering work in mobile computing targeted supporting mobility at the operating system level (e.g., the work on Coda [14]) or for specific application domains (e.g., repository-based in Bayou [34]). On the other hand, many commercial applications mask mobility by relying on proxy architectures, but assume the existence of a fixed infrastructure. Clearly, these approaches are limited in that they either solve issues that are specific for a given application domain, or do not address unconstrained mobile settings.

LIME (Linda in a Mobile Environment) [26, 18] is a model and a middleware expressly designed for supporting the development of mobile applications. In contrast with many of the existing proposals, LIME provides the ap-

*G.P. Picco (picco@elet.polimi.it) is with Dip. di Elettronica e Informazione, Politecnico di Milano, Italy. A.L. Murphy (murphy@cs.rochester.edu) is with Dept. of Computer Science, University of Rochester, NY, USA. G.-C. Roman (roman@cs.wustl.edu) is with Dept. of Computer Science and Engineering, Washington University in St. Louis, MO, USA.

plication programmer with a set of general-purpose programming constructs. Moreover, by adopting a peer-to-peer architecture that does not rely on any fixed infrastructure, it addresses the needs of the most radical forms of mobility, and in particular of MANETs.

The design of LIME is inspired by the realization that the problem of designing applications involving mobility can be regarded as a coordination problem [30], and that a fundamental issue to be tackled is the provision of good abstractions for dealing with, and exploiting, a dynamically changing context. Coordination is defined as a style of computing that emphasizes a high degree of decoupling among the computing components of an application. As initially proposed in Linda [9], this can be achieved by allowing independently developed components to share information stored in a globally accessible, persistent, content-addressable data structure, typically implemented as a centralized tuple space. A small set of operations enabling the insertion, removal, and copying of tuples provides a simple and uniform interface to the tuple space. Temporal decoupling is achieved by dropping the requirement that the communicating parties be present at the time the communication takes place and spatial decoupling is achieved by eliminating the need for components to be aware of each other's identity in order to communicate. A clean computational model, a high degree of decoupling, an abstract approach to communication, and a simple interface are the defining features of coordination technology.

LIME reinterprets Linda within the mobile scenario in an original way. Each mobile unit is permanently associated with a local tuple space, whose content is transiently shared with the content of similar tuple spaces attached to the other units within range. Hence, the tuple space used for coordination is no longer unique, global, and persistent—assumptions that macroscopically conflict with mobility. Instead, it is dynamically built out of the spaces contributed by the mobile units within range, and reflects the current configuration of the system. Transiently shared tuple spaces are the key to shielding the programmer from the complexity of the system configuration, while still providing an effective abstraction for handling communication among application components. In addition, LIME defines constructs providing increased expressiveness, by introducing the ability to react asynchronously to the presence of a tuple, and to control the placement of a tuple and its access within the global tuple space. Finally, since no assumption is made about the nature of the mobile units, the computational model naturally encompasses both physical mobility of hosts and logical mobility of agents. The computational model is embodied in a Java-based middleware, made available as open source [33], which has been successfully employed for developing mobile applications.

In this contribution, we present LIME by focusing on the model concepts and the programming and design techniques useful to the developer of mobile applications. Other available papers describe the design of the middleware [18] and the formal semantics of the model [19]. The document is organized as follows. Section 2 provides the minimal Linda background necessary to understand LIME. Section 3 contains an overview of the LIME model and of the application programming interface of the companion middleware. Section 4 walks through a case study application, and shows how its requirements are satisfied through a design exploiting LIME. Section 5 presents some middleware extensions to LIME that we built entirely as an application layer on top of the original middleware. Section 6 places LIME in the context of related work. Finally, Section 7 completes our contribution with some brief concluding remarks.

2 Linda in a Nutshell

In Linda, processes communicate through a shared *tuple space* that acts as a repository of elementary data structures, or *tuples*. A tuple space is a multiset of tuples that can be accessed concurrently by several processes. Each tuple is a sequence of typed fields, such as ⟨"foo", 9, 27.5⟩, and contains the information being communicated.

Tuples are added to a tuple space by performing an $\mathbf{out}(t)$ operation, and can be removed by executing $\mathbf{in}(p)$. Tuples are anonymous, thus their selection takes place through pattern matching on the tuple content. The argument p is often called a *template* or *pattern*, and its 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 $\mathbf{rd}(p)$ 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 becomes available. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives **inp** and **rdp**, called *probes*, which allow non-blocking access to the tuple space[1].

[1] Additionally, Linda implementations often include also an **eval** operation which provides dynamic process creation and enables deferred

Moreover, some variants of Linda (e.g., [32]) provide also *bulk operations*, which can be used to retrieve all matching tuples in one step. In LIME we provide a similar functionality through the **ing** and **rdg** operations, whose execution is asynchronous like in the case of probes[2].

3 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 form of decoupling is of paramount importance in a mobile setting, where the parties involved in communication change dynamically due to their migration or connectivity patterns. Moreover, the notion of tuple space provides a straightforward and intuitive abstraction for representing the computational context perceived by the communicating processes. On the other hand, decoupling is achieved thanks to the properties of the Linda tuple space, namely its global accessibility to all the processes, and its persistence—properties that are clearly hard if not impossible to maintain in a mobile environment.

LIME [26, 18] adapts Linda to the mobile environment in an original way, and provides a coordination layer that can be exploited successfully for designing applications that exhibit logical mobility, physical mobility, or both. In this section we present the LIME model, hand in hand with the description of how its constructs are made available through the application programming interface of the companion middleware. The reader interested in additional programming details can find extensive documentation and examples on the LIME Web site [33].

3.1 Model Setting and Overview

The fundamental entities in LIME are *agents*, *hosts*, and *tuple spaces*. Agents are the only active components. Hosts can be mobile, and are mainly roaming containers which provide connectivity and execution support for agents. Agents can be mobile as well, and migrate across hosts of their own volition. The *connectivity* patterns among agents and hosts constrain coordination in LIME. Mobile hosts are connected when a communication link is available. Mobile agents are connected when they are co-located on the same host, or they reside on hosts that are connected. Changes in connectivity among hosts depend only on changes in the physical communication links. Connectivity among mobile agents may depend also on arrival and departure of agents, with creation and termination of mobile agents being regarded as special cases of connection and disconnection, respectively.

Tuple spaces provide the coordination media among agents, as in Linda. Nevertheless, in LIME each agent is permanently associated with at least one *interface tuple space* (ITS). This tuple space constitutes the only access to the data context for the agent it is associated with and, at the same time, it contains the tuples the agent is willing to make available to the rest of the system. At any time, an agent can access through its ITS the union of the content of the ITS of all the agents currently connected. In essence, we shift from the fixed context characteristic of Linda to a dynamically changing one by breaking up the single global tuple space into many, and by introducing rules for transient sharing of these individual tuple spaces based on connectivity. The expressive power of the model is then increased further by the ability to execute reactive and asynchronous operations, and to restrict the scope of operations based on location. We now describe in more detail the features of the LIME model and middleware.

3.2 Creating a LIME Tuple Space

Figure 1 shows the public interface[3] of the class `LimeTupleSpace`, which embodies the concept of a LIME transiently shared tuple space. The association between an agent and the tuple space is established at creation time, by invoking the constructor. Agents may have multiple ITSs distinguished by a name since this is recognized [6] as a useful abstraction to separate related application data. The name of the tuple space is specified as a parameter of the constructor. In the current implementation, agents are single-threaded and only the thread of the agent that creates the tuple space is allowed to perform operations on the `LimeTupleSpace` object; accesses by other threads fail by returning an exception. This represents the constraint that the ITS must be permanently and exclusively attached to the corresponding mobile agent.

evaluation of tuple fields. For the purposes of this work, however, we do not consider this operation further.

[2]Hereafter we often do not mention this pair of operations, since they are useful in practice but do not add significant complexity either to the model or to the implementation.

[3]Exceptions are not shown for the sake of readability.

```
public class LimeTupleSpace {
  public LimeTupleSpace(String name);
  public String getName();
  public boolean isOwner();
  public boolean isShared();
  public boolean setShared(boolean isShared);
  public static boolean setShared(LimeTupleSpace[] lts, boolean isShared);
  public void out(ITuple tuple);
  public ITuple in(ITuple template);
  public ITuple rd(ITuple template);
  public void out(AgentLocation destination, ITuple tuple);
  public ITuple in(Location current, AgentLocation destination, ITuple template);
  public ITuple inp(Location current, AgentLocation destination, ITuple template);
  public ITuple[] ing(Location current, AgentLocation destination, ITuple template);
  public ITuple rd(Location current, AgentLocation destination, ITuple template);
  public ITuple rdp(Location current, AgentLocation destination, ITuple template);
  public ITuple[] rdg(Location current, AgentLocation destination, ITuple template);
  public RegisteredReaction[] addStrongReaction(LocalizedReaction[] reactions);
  public RegisteredReaction[] addWeakReaction(Reaction[] reactions);
  public void removeReaction(RegisteredReaction[] reactions);
  public boolean isRegisteredReaction(RegisteredReaction reaction);
  public RegisteredReaction[] getRegisteredReactions();
}
```

Figure 1: The class `LimeTupleSpace`, representing a transiently shared tuple space.

When the ITS is first created and bound to an agent, its sharing status is *private*, meaning that the only agent that can access the ITS's data is the one associated with it at creation time. A private ITS can be used as a stepping stone to a shared data space, allowing the agent to populate it with data prior to making it publicly accessible. Alternately, it can be useful as a primitive data structure for local data storage. LIME operations, shown in the remainder of Figure 1 and discussed in the rest of this section, are available also on a private tuple space. For instance, the **inp** operation is provided through the method `inp`, which accepts a parameter representing the template to be matched against, and returns a matching tuple, if any, or `null`. The only requirement for tuple objects is to implement the interface `ITuple`, which is defined in a separate package called LIGHTS [23] that provides access to a lightweight tuple space implementation. Note also that **inp**, and similarly the other operations, are overloaded with additional parameters representing locations, whose meaning is discussed in the following.

One relevant difference, however, is that blocking operations are forbidden on a private tuple space, and return an exception. In fact, since the private tuple space is exclusively associated to one agent, the execution of an **in** or **rd** when no matching tuple is present would suspend the agent forever, effectively waiting for a tuple that no other agent can possibly insert.

3.3 Enabling Transient Sharing

When an agent is alone in the system, i.e., there are no other agents co-located on the same machine or on machines in range, the ITS of the agent is the only available data repository accessible by it. On the surface, this could seem a situation identical to an agent owning a private tuple space, but the two cases are actually rather different. Irrespective of the system configuration, a private tuple space is never shared, and its existence is known only to the agent that owns it. Instead, a non-private tuple space is available for sharing with other units, according to connectivity.

When multiple mobile units[4] are able to communicate, either directly or transitively, we say these units form a LIME *group*. We can restrict the notion of group membership beyond simple communication, but for the purposes of this document, we consider only connectivity. The semantics of LIME is such that the content of the ITSs of all group members are merged, or transiently shared, to form a single, large context that is accessed by each unit through its own ITS. The sharing itself is transparent to each mobile unit, however as the members of the group change, the content of the tuple space each member perceives through operations on the ITS changes in a transparent way. The joining of a group by a mobile unit, and the subsequent merging of its local context with the group context is referred to as *engagement*, and is performed as a single, atomic operation. A mobile unit leaving a group triggers *disengagement*, that is, the atomic removal of the tuples representing its local context from the remaining group context. In general, whole groups can merge, and a group can split into several groups due to changes in connectivity.

In the case of multiple tuple spaces associated to agents, the sharing rule relies on tuple space names: only identically-named tuple spaces are transiently shared among the members of a group. Thus, for instance, when an

[4]In the following, we use the term *unit* when we want to refer to agents and hosts, without making a distinction.

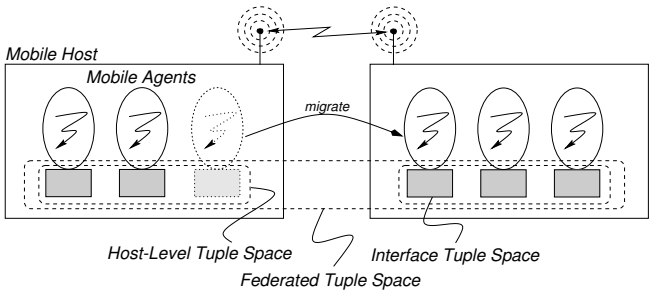

Figure 2: Transiently shared tuple spaces encompass physical and logical mobility.

agent a owning a single tuple space named X joins a group consisting of one single agent b that owns two tuple spaces named X and Y, only X becomes shared between the two agents. Tuple space Y remains accessible only to b, and potentially to other agents owning Y that may join the group later on.

Since all tuple spaces are initially created private, sharing must be explicitly enabled by calling the instance method `setShared`, shown in Figure 1. The method accepts a boolean parameter specifying whether the transition is from private to shared (`true`) or vice versa (`false`). Calling this method effectively triggers engagement or disengagement of the corresponding tuple space. The sharing status can also be changed in a single atomic step for multiple tuple spaces owned by the same agent by using the `static` version of `setShared` (see Figure 1). Engagement or disengagement of an entire host, instead, can be triggered explicitly by the programmer by using the methods `engage` and `disengage`, provided by the `LimeServer` class, not shown here. Otherwise, they are implicitly called by the run-time support according to connectivity. The `LimeServer` class is essentially an interface to the run-time support that exports additional system-related features, e.g., loading of an agent into a local or remote run-time support, setting of properties, and so on. In particular, it also allows the programmer to limit transient sharing to the tuple spaces residing on the host, instead of spanning the whole system.

3.4 Reconciling Different Forms of Mobility

One of the key features of LIME is the ability to encompass both physical mobility of hosts and logical mobility of agents in a single coordination framework. The relationship between the two forms of mobility is illustrated in Figure 2. The transiently shared ITSs belonging to the agents co-located on a host define a *host-level tuple space*. This, in turn, can be regarded as the ITS associated to the host, and hence transient sharing determines a *federated tuple space*. When a federated tuple space is established, a query on the shared ITS of an agent returns a tuple that may belong to the tuple space carried by that agent, to a tuple space belonging to a co-located agent, or to a tuple space associated with an agent residing on some remote, connected host.

Although LIME provides a uniform treatment of these two forms of mobility, it is worth making some observations. First, many applications do not need both forms of mobility. However, straightforward adaptations of the model are possible. For instance, applications that do not exploit mobile agents but run on a mobile host can employ one or more stationary agents, i.e., programs that do not contain migration operations. In this case, the design of the application can be modeled in terms of mobile hosts whose ITS is a fixed host-level tuple space. Instead, applications that do not exploit physical mobility—and do not need a federated tuple space spanning different hosts—can exploit only the host-level tuple space as a local communication mechanism among co-located agents.

Second, it is interesting to note how mobility is not dealt with directly in LIME, i.e., there are no constructs for triggering the movement of agents or hosts. Instead, the effect of migration is made indirectly manifest in the model and middleware only through changes observed in the connectivity among components. This choice, that sets the nature of mobility aside, keeps our model general and enables different instantiations of the model based on different notions of connectivity. This choice is retained also at the middleware level, where agent mobility is not supported directly. Instead, as with tuple spaces, agent migration is decoupled from the rest of the system by an adaptation layer that simplifies the integration of a mobile agent system. The currently available implementation relies on an adaptor built for the μCODE mobile code toolkit, developed by one of the authors [24] and available as open source [22]. This adaptation layer allows a mobile agent to carry along one or more LIME tuple spaces, and automatically deals with their (dis)engagement. Upon a migration, the agent tuple spaces are all toggled to private, and hence disengaged. These tuple spaces are serialized as part of the agent state and migrated to the destination along with the agent, where

Current location	Destination location	Defined projection
unspecified	unspecified	Entire federated tuple space
unspecified	λ	Tuples in the federated tuple space and destined to λ
ω	unspecified	Tuples in ω's tuple space
Ω	unspecified	Tuples in Ω's host-level tuple space, i.e., belonging to any agent at Ω
ω	λ	Tuples in ω's tuple space and destined to λ
Ω	λ	Tuples in Ω's host-level tuple space and destined to λ

Table 1: Accessing different portions of the federated tuple space by using location parameters. In the table, ω and λ are agent identifiers, while Ω is a host identifier.

they are deserialized and shared again before the agent code begins to execute. More details about the adaptation layer and how to integrate a mobile agent system with LIME are available in the LIME documentation, and on the LIME Web site [33].

3.5 Restricting the Scope of Operations

Transiently shared tuple spaces foster a style of coordination that reduces the details of distribution and mobility to content changes in what is perceived as a local tuple space. This view is very powerful, and greatly simplifies application design in many scenarios by relieving the designer from the chore of maintaining explicitly a view of the context consistent with changes in the configuration of the system. On the other hand, this view may hide too much in situations where the designer needs more fine-grained control over the portion of context that must be accessed. For instance, the application may require control over the agent responsible for holding a given tuple, something that cannot be specified only in terms of the global context. Also, performance and efficiency considerations may come into play, as in the case where application information would enable access aimed at a specific host-level tuple space, thus avoiding the greater overhead of a query spanning the whole federated tuple space. Such fine-grained control over the context perceived by the mobile unit is provided in LIME by extending the Linda operations with tuple location parameters that operate on user-defined projections of the transiently shared tuple space. Further, all tuples are implicitly augmented with two fields, not directly accessible to the programmer, representing the tuple's *current* and *destination location*. The current location identifies the single agent responsible for holding the tuple when all agents are disconnected, and the destination location indicates the agent with whom the tuple should eventually reside.

The $\mathbf{out}[\lambda]$ operation extends **out** with a location parameter representing the identifier of the agent responsible for holding the tuple. The semantics of $\mathbf{out}[\lambda](t)$ involve two steps. The first step is equivalent to a conventional $\mathbf{out}(t)$, whose effect is to insert the tuple t in the ITS of the agent calling the operation, say ω. At this point t has a current location ω, and a destination location λ. If the agent λ is currently connected, the operation is completed as a single atomic step by moving the tuple t to the destination location. On the other hand, if λ is currently disconnected the tuple remains at the current location, the tuple space of ω. This *"misplaced"* tuple, if not withdrawn[5], will remain misplaced unless λ becomes connected. In this case, the tuple will migrate to the tuple space associated with λ as part of the engagement. By using $\mathbf{out}[\lambda]$, the caller can specify that the tuple is supposed to be placed within the ITS 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.

Variants of the **in** and **rd** operations using location parameters are allowed as well. These operations, of the form $\mathbf{in}[\omega, \lambda](p)$ and $\mathbf{rd}[\omega, \lambda](p)$, enable the programmer to refer to a projection of the current context defined by the value of the location parameters, as illustrated in Table 1. The current location parameter enables the restriction of scope from the entire federated tuple space (no value specified) to the tuple space associated to a given host or even a given agent. The destination location is used to identify misplaced tuples.

Location parameters are specified in the middleware by using the classes `AgentLocation` and `HostLocation`, both subclasses of `Location`. These classes enable the definition of globally unique location identifiers for hosts and agents, and are used to specify different scopes for LIME operations. For instance, a probe `inp(cur,dest,t)` may be restricted to the tuple space of a single agent if `cur` is of type `AgentLocation`, or it may refer the whole host-

[5]Specifying a destination location λ implies neither guaranteed delivery nor ownership of the tuple t to λ. Linda rules for non-deterministic selection of tuples are still in place; thus, it might be the case that some other agent may withdraw t from the tuple space before λ, even after t reached λ's ITS.

level tuple space, if `cur` is of type `HostLocation`, according to Table 1. The constant `Location.UNSPECIFIED` is used to allow any location parameter to match. For instance, `in(cur,Location.UNSPECIFIED,t)` returns a tuple contained in the tuple space of `cur`, regardless of its final destination, including also misplaced tuples. Note how typing rules allow the proper constraint of the current and destination locations according to the rules of the LIME model. For instance, the `destination` parameter is always an `AgentLocation` object, as agents are the only carriers of tuples in LIME. In the current implementation, probes are always restricted to a local subset of the federated tuple space, as defined by the location parameters. An unconstrained definition, as the one provided for `in` and `rd`, would involve a distributed transaction in order to preserve the semantics of the probe across the federated tuple space.

3.6 Reacting to Changes

LIME extends Linda not only by providing transiently shared tuple spaces, but also by introducing the notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. Informally[6], a reaction can *fire* if a tuple matching pattern p exists in the tuple space. After every regular tuple space operation, a reaction is selected non-deterministically and, if it is enabled, the statements in s are executed in a single, atomic step. This selection and execution continues until no reactions are enabled, at which point normal processing resumes. Blocking operations are not allowed in s, as they may prevent the execution of s from terminating.

Reactions provide the programmer with very powerful constructs for specifying the actions that need to take place in response to a *state* change, and ensure their execution in a single atomic step. In particular, it is worth noting how this model is much more powerful than many event-based ones [31], including those exploited by tuple space middleware such as TSpaces [12] and JavaSpaces [13], which are typically stateless and provide no guarantee about the atomicity of event reactions.

Nevertheless, this expressive power comes at a price, especially in a distributed setting. When multiple hosts are present, the content of the federated tuple space is scattered among several agents. Thus, maintaining the requirements of atomicity and serialization imposed by reactive statements requires a distributed transaction encompassing several hosts—very often, an impractical solution. For specific applications and scenarios, e.g., those involving a very limited number of hosts, or those exploiting only local interactions among mobile agents, these kind of reactions, referred to as *strong reactions*, are still reasonable. For practical performance reasons, however, our implementation currently limits the use of strong reactions by restricting the current location field to be a host or agent, and by enabling a reaction to fire only when the matching tuple appears on the same host as the agent that registered the reaction. As a consequence, a mobile agent can register a reaction for a host different from the one where it is residing, but such a reaction remains disabled until the agent migrates to the specified host. These constraints effectively force the *detection* of a tuple matching p and the corresponding *execution* of the code fragment s to take place (atomically) on a single host, and hence does not require a distributed transaction.

To strike a compromise between the expressive power of reactions and the practical implementation concerns we introduced another reactive construct that allows some form of reactivity spanning the whole federated tuple space but with weaker semantics. The processing of a *weak reaction* proceeds as in the case of a strong reaction, but detection and execution do not happen atomically: instead, execution is guaranteed to take place only eventually, after a matching tuple is detected. The execution of s takes place on the host of the agent that registered the reaction.

The use of reactions involves the operations in `LimeTupleSpace` and the classes shown in Figure 3. Reactions can be registered on a tuple space by invoking either `addStrongReaction` or `addWeakReaction`. These methods return an object `RegisteredReaction`, which can be used to deregister a reaction with the method `removeReaction` and provides additional information about the registration process. The decoupling between the reaction used for the registration and the `RegisteredReaction` object returned allows for registration of the same reaction on different ITSs and for the same reaction to be registered with strong and, subsequently, with weak semantics.

Reactions can be annotated with location parameters, with the same meaning discussed earlier for **in** and **rd** and shown in Table 1. Reactions can be either of type `LocalizedReaction`, where the current and destination location restrict the scope of the operation, or `UbiquitousReaction`, that specifies the whole federated tuple space as a target for matching. In the current implementation strong reactions are confined to a single host, and hence only a `LocalizedReaction` can be passed to `addStrongReaction`. The reaction type is used to enforce the proper

[6]The semantics of reactions are based on the Mobile UNITY reactive statements [16]. The reader interested in formal details is redirected to [19].

```
public abstract class Reaction {
  public final static short ONCE;
  public final static short ONCEPERTUPLE;
  public ITuple getTemplate();
  public ReactionListener getListener();
  public short getMode();
  public Location getCurrentLocation();
  public AgentLocation getDestinationLocation();
}
public class UbiquitousReaction extends Reaction {
  public UbiquitousReaction(ITuple template, ReactionListener listener, short mode);
}
public class LocalizedReaction extends Reaction {
  public LocalizedReaction(Location current, AgentLocation destination,
                           ITuple template,ReactionListener listener, short mode);
}
public class RegisteredReaction extends Reaction {
  public String getTupleSpaceName();
  public AgentID getSubscriber();
  public boolean isWeakReaction();
}
public class ReactionEvent extends java.util.EventObject {
  public ITuple getEventTuple();
  public RegisteredReaction getReaction();
  public AgentID getSourceAgent();
}
public interface ReactionListener extends java.util.EventListener {
  public void reactsTo(ReactionEvent e);
}
```

Figure 3: The classes `Reaction`, `RegisteredReaction`, `ReactionEvent`, and the interface `ReactionListener`, required for the definition of reactions on the tuple space.

registration constraint through type checking. The common ancestor class `Reaction` defines a number of accessors for the properties established for the reaction at creation time. Creation of a reaction is performed by specifying the template that needs to be matched in the tuple space, a `ReactionListener` object that specifies the actions taken when the reaction fires, and a reaction *mode* that controls 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. The `ReactionListener` interface requires the implementation of a single method `reactsTo` that is invoked by the run-time support when the reaction actually fires. This method has access to the information about the reaction carried by the `ReactionEvent` object passed as a parameter to the method.

3.7 Accessing the System Configuration

Thus far, our extension of Linda operations with location parameters hides completely the details of the system (re)configuration that generated those changes. For instance, if the probe $\mathbf{inp}[\omega, \lambda](p)$ fails, this simply means that no tuple matching p is available in the projection of the federated tuple space defined by the location parameters $[\omega, \lambda]$. It cannot be directly inferred whether the failure is due to the fact that agent ω does not have a matching tuple, or simply agent ω is currently not part of the group.

Without awareness of the system configuration, only a partial context awareness can be accomplished, where applications are aware of changes only in the portion of context concerned with application data. Although this perspective is often enough for mobile applications, in many others the portion of context more closely related to the system configuration plays a key role. For instance, in some circumstances it becomes necessary to react to the departure of a mobile unit, or to determine the set of units currently belonging to a LIME group. Interestingly, LIME provides this form of awareness of the system configuration by using the same abstractions discussed thus far: 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 in the group and their relationship, e.g., which agents are being supported by which host, or which tuple spaces are being shared by which agent. Insertion and withdrawal of tuples in LimeSystem is a prerogative of the run-time support. Nevertheless, applications can read tuples and register reactions to respond to changes in the configuration of the system.

3.8 Implementation Details

The `lime` package is roughly 5,000 non-commented source statements, resulting in an approximately 100 Kbyte `jar` file. The LIGHTS lightweight tuple space implementation and the adapter for integrating multiple tuple space engines adds an additional 20 Kbyte `jar` file. When using mobile agents, the μCODE toolkit adds approximately 30 Kbyte in a `jar` file. Communication is completely handled at the socket level, requiring no support for RMI or other communication mechanisms. A new version of LIME exploits Global Positioning System (GPS) to automatically trigger engagement and disengagement based on physical position, along the lines of the algorithm described in [29]. Thus far, LIME has been tested successfully on PCs running various versions of Windows and Linux, and exploiting both wired Ethernet as well as IEEE 802.11 wireless technology. Moreover, LIME runs successfully on PDAs equipped with PersonalJava.

4 Application Example

LIME has been successfully applied in the development of several applications in the mobile environment. In this section we present a single application, a jigsaw assembly game, throughout the development process from requirements to implementation, showing the thought process applied when designing mobile applications over LIME.

4.1 Requirements

The goal is to build a jigsaw assembly game for multiple players in the mobile ad hoc environment. The game should reasonably emulate the physical world process of assembling a jigsaw puzzle where an individual player starts a puzzle by dumping the pieces out of the box into a common area. Other players join, and the puzzle is assembled through the joint effort of the individual players.

When considering this process in a mobile environment in which each player is equipped with a palm- or lap-top computer, the following requirements must be met. First, players who are currently connected should be able to see the piece assemblies of one another as soon as possible. In other words, if one player, p_1, assembles two puzzle pieces on her laptop and another player, p_2, is connected, p_2's display should be updated quickly to show that the pieces have been assembled. Second, as this is a mobile game and the players are not expected to remain connected for the duration of the puzzle assembly, it should be possible for a player to make assemblies of pieces while disconnected. This leads to the next requirement, namely that when two previously disconnected players reconnect, their displays should be updated to show the changes made by one another. Finally, the game should be able to support multiple, concurrent puzzles, and a single player should be able to participate in more than one puzzle at a time.

4.2 Design and Implementation

The requirements sketched above are intentionally vague, leaving many options available. The design and implementation take into consideration the programming style encouraged by LIME as well as the constraints of the wireless, mobile environment. Here we describe the puzzle application, originally assigned as a course project, outlining first the design choices for the use of the tuple spaces and tuples, followed by the implementation of the user actions, and finishing with the updating of the user interface.

Tuple spaces and tuples. The first choices in the design of all LIME applications are the use of the federated tuple space(s) and the format of the tuples. In order to support multiple concurrent puzzles, an obvious choice is to use a separate tuple space for each puzzle. This effectively separates the actions and pieces of distinct puzzles, and easily allows a player to participate in as many puzzles as she would like. When a player chooses to start a puzzle, she must provide a name to be used as the tuple space name. Since the names of all active puzzles are present in the LimeSystem tuple space, the puzzle application can query this space and display the available games to the user. Joining a puzzle is equivalent to creating a tuple space with its name.

Since each tuple space is used as the repository for the current state of a single puzzle, the next choice is how to represent this information in tuples. This is achieved with two kinds of tuples: *image* and *assembly*. An image tuple exists for every puzzle piece and contains two fields: the identifier and the bitmap of the piece. The second kind of tuple, the assembly, represents a group of connected puzzle pieces. Each such tuple contains a single field with the list of the identifiers of the connected pieces. When a new game starts, for each puzzle piece two tuples are inserted

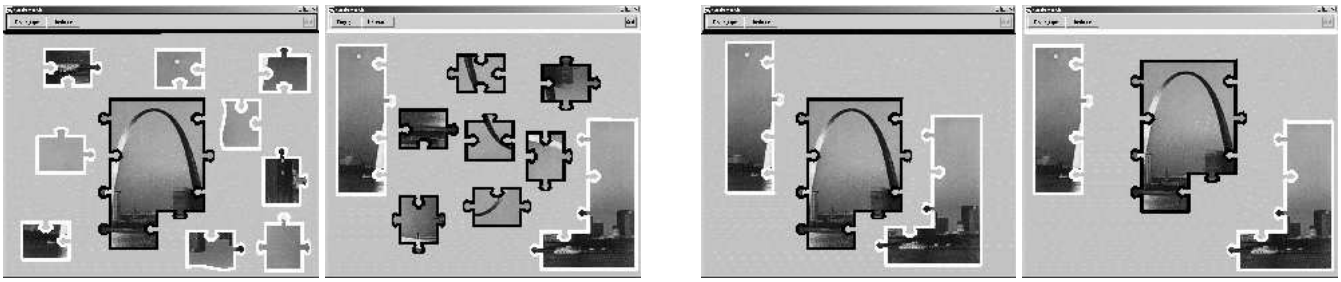

Figure 4: Jigsaw assembly game. The left two images show the puzzle trays of the black and white players while they are disconnected and able to assemble only their selected pieces. The right two images show the black and white puzzle trays after the players re-engage and see the assemblies that occurred during disconnection.

with a corresponding **out** operation: one contains the image and one contains the representation of an assembly with a single piece, i.e., a list whose single element is the piece identifier. When two pieces are assembled, the two assembly tuples of the original pieces are replaced by a single tuple representing the change.

There are two important benefits of our choice for representing the puzzle data, thanks to the fact that, when puzzle pieces are assembled, image tuples do not change. Since these tuples contain bitmaps that are likely to be large in comparison to the assembly tuples, we save the computation time necessary to remove and reinsert the large image tuple each time an assembly is made, and we also save bandwidth as the images are not repeatedly transmitted over the wireless link.

User actions. The next decision is where the tuples should reside throughout the federated tuple space. It is clear that when all players are connected, all the tuples representing the puzzle, both images and assemblies, are present. However, when the players are disconnected, the puzzle pieces are divided among the tuple spaces of the players, and therefore are not accessible to everyone. This brings us to the first of the user operations, namely the ability to select a piece and become its "owner". When a player owns a piece, that piece resides in her portion of the federated tuple space, and the piece remains with the player even after disconnection. This notion of ownership is on the assembly level, but must also extend to images. In other words, when a player takes ownership of an assembly, it must also take ownership of the images associated with the pieces of the assembly.

At the user interface level, selection of a piece is achieved by right clicking on one of the pieces displayed on the screen. To show the player which pieces have been selected by which players, we associate a color with each player, and outline the selected pieces with this color, as shown in Figure 4. In LIME, selection is accomplished by performing **inp** operations to retrieve both the image and assembly tuples, followed by **out** operations to reinsert the tuples into the tuple space. Because we do not specify a destination location for the **out** operations, the default assigns tuples a current field equal to the new player. Since the **inp** operation must specify the (current) location from which to retrieve the tuples, this information is stored with each puzzle piece appearing on the display.

This choice to have pieces belong to players allows a player to assemble pieces while disconnected. However, it implies that a player should only be allowed to assemble pieces that she owns. It also prevents two disconnected players from using the same piece in two different assemblies. This maintains the consistency of the puzzle, despite disconnections. One can argue that because there is only one correct way to assemble a puzzle, the use of a piece in more than one assembly can easily be resolved upon reconnection of the players, thus our choice is overly restrictive. However, for the sake of the example, we have chosen to model an application where such concurrent changes are not permitted.

Piece assembly, the core of the game, is similar to selection. First, the two assembly tuples representing the pieces are removed from the player's tuple space with **inp** operations, then the new assembly tuple is written with an **out** operation. Nevertheless, concurrency issues become relevant at this point. Consider the case where player p_1 is trying to assemble pieces t_1 and t_2 while, at the same time, player p_2 is trying to select piece t_2 from p_1's tuple space. The following sequence of actions may take place: p_1 successfully issues the first **inp** removing t_1, p_2 successfully removes t_2 and hence becomes its owner, and finally p_1 executes its second **inp**, which at this point is bound to return `null`. In this case, p_1 cannot complete successfully the assembly, and must reverse the assembly process by reinserting t_1 into the tuple space, thereby retaining consistency of the puzzle state. To the user, we indicate this failure with an audible beep.

Display update. The main display for each player is one *puzzle tray* for each puzzle she is participating in. The requirements state that the puzzle tray must be kept up to date as changes are made. Unlike the previous operations which are performed at the user's request, the updating of the puzzle tray is done in response to changes, and thus is most naturally implemented with a LIME reaction. Specifically, we use a *single* ONCEPERTUPLE, ubiquitous reaction on the federated tuple space, registered for assembly tuples. When this reaction fires, the list of pieces in the assembly tuple, contained in the `ReactionEvent` object passed to the listener code, is examined. If these pieces are already in the puzzle tray, they are rearranged to reflect the connection contained in the assembly list. If they are not in the tray, the images of the pieces are retrieved with **rdp** operations, and the puzzle tray is updated. LIME requires that **rdp** operations specify the current location of the tuples. In this case, the reaction contains the source of the assembly tuple, and this value is used to retrieve the image tuples.

This single reaction, installed at each player's client, updates the screen in all cases including the initiation of a new puzzle, piece selection, assembly, and updating the puzzle tray upon reconnection. When a new puzzle is started, this reaction fires once for every assembly tuple, the **rdp** is executed to retrieve the image tuples, and the puzzle tray is populated with the puzzle pieces. When a player selects a piece, the **out** operation that reinserts the assembly tuple causes the reaction to fire. This time, because the graphic for the tuple has already been displayed, the image tuple is not retrieved, but the screen is updated to reflect the change in the outline color of the puzzle piece. Assembly of pieces similarly creates a new assembly tuple that is reacted to, updating the display accordingly. Finally, when two players reconnect after a disconnection, the requirements state that the puzzle trays of the players must be updated to reflect the changes made during the disconnection. Because these changes are represented in assembly tuples that are new to the previously disconnected player, the reaction fires, and the display is updated. Figure 4 shows the appearance of the puzzle tray during disconnection and after reconnection.

It is interesting to note how a huge fraction of the application behavior is handled through a single reaction. The ability to specify asynchronous state-based reactive behavior and declaratively specify the whole system as its scope, together with the notion of transiently shared tuple spaces, greatly simplifies the programmer's task. Indeed, a look at the source code reveals that a great deal of the programming effort was devoted to proper implementation of the user interface, with only a small percentage of the code devoted to managing distribution and mobility with LIME.

4.3 Beyond the Puzzle

From the description, it is evident that our jigsaw assembly game embodies a pattern of interaction where the shared workspace displayed by the user interface of each player provides an accurate image of the state of all connected players, but only a weakly consistent image of the global state of the system. For instance, a user's display contains only the last known information about each puzzle piece in the tray. If two pieces have been assembled by a disconnected player, this change is not visible to others. However, this still allows the players to work towards achieving the global goal, i.e., the solution of the puzzle, through incremental updates of their local state.

This application is a simple game that nonetheless exhibits the characteristics of a general class of applications in which data sharing is the key element. Hence, the design strategy we exploited here may be adapted easily to handle updates in the data being shared by real applications. One example could be collaborative work applications involving mobile users, where our mechanism could be used to deal with editorial changes in sections of a document, or with paper submissions and reviews evaluated by a program committee.

Other applications exhibiting diverse patterns of interaction are available, in source code form, at `lime.sf.net`.

5 Building Middleware Functionality on Top of LIME

In designing LIME we strived for minimality, in an attempt to identify a core of concepts and constructs general enough to be used as building blocks for higher-level services, and yet powerful enough to satisfy the basic needs of most mobile applications. Application development with LIME, an example of which we described in the previous section, gave us the opportunity to evaluate the expressive power of LIME constructs in building mobile applications. In this section we report on experiences that show how LIME can be used effectively also to build high-level middleware services that, nonetheless, do not require modifications to the original middleware.

5.1 Transiently Shared Code Bases

In our description of LIME, we always implicitly assumed that a LIME tuple space contains data. Instead, in the work described in [25] we explored the opportunities opened by storing *code* in a LIME tuple space, while still exploiting its transient sharing and reactive features. While the idea is very simple, its implications are far reaching, and hold the potential to change fundamentally the mechanisms usually exploited for supporting mobility of code.

Currently available support for mobile code is mostly limited to variations of a code on demand approach [8] where the code is dynamically downloaded from a well-known site at name resolution time. Examples are Java applets in Web browsers and dynamic downloading of stubs in Java/RMI and Jini. Unfortunately, in its most common incarnations this approach has at least two relevant drawbacks. First of all, the local *code base*, i.e., the set of classes locally available, is usually accessible only to the run-time support, and hence it remains hidden from the applications. This prevents the development of code caching schemes with application-level policies, e.g., to intelligently cache or discard code on resource-constrained devices. Moreover, remote dynamic linking usually relies on a well-known centralized code base. This scheme evidently breaks when applied in a fluid scenario like the one defined by MANETs, but has drawbacks also in a fixed scenario, since it does not exploit the potential presence of suitable code on nearby hosts.

Using LIME tuple spaces to store code changes the situation dramatically. An agent can now manipulate its own code base using LIME primitives. Moreover, since each tuple space is permanently and exclusively associated with its agent, when the latter moves its code base migrates along with it. Finally, transient sharing effectively stretches the boundaries of an agent code base to an extent possibly covering the whole system at hand. These characteristics provide an elegant solution to the problem we mentioned earlier. A proper redefinition of the class loader, like the one described in [25], can operate on the LIME tuple space associated with the agent for which the class needs to be resolved and query it using the operations provided by LIME. Thus, the class loading mechanism can now resolve class names by leveraging off the federated code base to retrieve and dynamically link classes in a location transparent fashion (e.g., through a **rd**) or use location parameters to narrow the scope of searches (e.g, down to a given host or agent).

Nevertheless, the use of transiently shared tuple spaces need not be confined to the innards of the class loading mechanism, rather agents can be empowered with the ability to manipulate directly the federated code base. Hence, not only can an agent proactively query up to the whole system for a given class, but it can also insert a class tuple into the code base of another agent by using the $\mathbf{out}[\lambda]$ operation, with the semantics of engagement and misplaced tuples taking care of disconnection and subsequent reconciliation of the federated code base. This new class can then be used by the receiving agent to execute tasks in previously unknown ways, or behave according to a new coordination protocol. Blocking operations acquire new uses, allowing agents to synchronize not only on the presence of data needed by the computation, but also on the presence of code needed to perform, or augment, the computation itself. LIME reactive operations provide additional degrees of freedom, by allowing agents to monitor the federated code base and react to changes with different atomicity guarantees. Reactions can be exploited to monitor the federated code base for new versions of relevant classes. Replication schemes can be implemented where a new class in an agent's code base is immediately replicated into the code base of all the other agents. The content of an agent's code base can be monitored to be aware of the current "skills" of the agent. The possibilities become endless.

Essentially, by exploiting the notion of transiently shared tuple space for code mobility we defined an enhanced coordination approach that, besides accommodating reconfiguration due to mobility and providing various degrees of location transparency, enables a new form of coordination no longer limited to data exchange, but encompassing also the exchange of fragments of behavior.

5.2 Service Provision

LIME's flexible support for application development over ad hoc networks received renewed validation as we considered the issue of service provision, an area in which the client-server model continues to dominate. Central to supporting service provision is the notion of discovering services at run time by relying on the service registration and discovery mechanisms. LIME made it possible to offer a solution that entails a new kind of service model built as a simple adaptation layer. The resulting veneer [10] uses LIME tuple spaces to store service advertisements and pattern matching to find services of interest and exploits the transient tuple space sharing feature of LIME to provide consistent views of the available services. The resulting system completely eliminates network awareness from the process of service discovery and utilization. The client only has to ask for the service it needs and does not have to know how the service will be reached. Furthermore, the model provides a distributed service registry that is

guaranteed to reflect the real availability of services at every moment in a mobile ad hoc environment. Consistent representation of service availability is obtained by atomically updating the view of the service repository as new connections are established or existing ones break down.

At the implementation level, a Jini-like interface [17] provides primitives for service advertisement and lookup. Every agent employs a tuple space to hold its own service registry where it advertises the services it provides. Advertisements may include proxies offering a service interface and encapsulating the communication mechanisms; the latter can be done in a manner that accommodates the mobility of both service providers and clients. As agents and hosts move, the registries of co-located agents are automatically shared. Thus, an agent requesting a service that is provided by a co-located agent can always access the service. If two hosts are within communication range they form a community and their service registries engage, forming a federated service registry. Upon engagement, the primitives operating on the local service registry are extended automatically to the entire set of service registries present in the ad hoc network. The sharing of the service registries is completely transparent to agents as agents in the community access the federated registry via their own local registries. The reliance on LIME concepts allowed for the fast deployment of the new service infrastructure specialized for mobile ad hoc settings with minimal programming effort. Later efforts [11] built upon this result to add secure service provision to the system by protecting tuple spaces with passwords and by using the same passwords to generate keys used to encrypt wireless traffic involving tuple spaces in general and federated registries in particular.

6 Related Work

A number of models and systems developed for either physical or logical mobility exhibit ideas that are somewhat similar to those put forth by LIME. Nevertheless, the concept of transiently shared tuple spaces and the semantics of reactions are unique to LIME. It is also the first system to explicitly address mobile ad hoc networks and to reunite physical and logical mobility under a common coordination framework.

Distributed Linda implementations have been studied extensively, but mostly with the goal of providing fault tolerance [36, 1] and data availability [28]. These systems typically exploit replication, as opposed to transient sharing, and assume a high degree of connectivity among the nodes hosting the distributed portions of the tuple space, a property that hampers their direct use in the mobile environment.

Recent years have seen a revitalization of Linda, also from a commercial point of view. Sun and IBM have developed their tuple space implementations for client-server coordination, i.e., JavaSpaces [13] and TSpaces [12], respectively. These systems present a centralized tuple space, accessible by remote clients. It is often claimed that these systems support mobility. This is true, in that they provide the equivalent of a proxy architecture. However, as we discussed earlier, this architecture exhibits a high-degree of centralization, and is inappropriate for the full fledged mobility of ad hoc networks.

The only Linda-like system explicitly supporting physical mobility we are aware of is Limbo [2, 35]. In this system, however, the emphasis is not on providing a general-purpose programming platform for mobile computing, but on providing network-level quality of service. The information necessary to this end is stored in dedicated tuple spaces on the mobile hosts, and can be made remotely accessible to agents sitting on different hosts. Interestingly, the Limbo *universal tuple space*, serving as a registry for all tuple spaces, is similar to LIME's LimeSystem tuple space. However, instead of describing the *current* system context, the universal tuple space remembers all tuple spaces the host has ever encountered irrespective of the current connectivity. In general, while Limbo tuple spaces may span multiple hosts, the mechanisms governing distribution (e.g., relocation of tuples) are unclear. Moreover, no form of reaction or event notification is provided.

As for logical mobility, a number of models are inspired by Linda. Nevertheless, in these models the tuple space is always exploited as a data repository explicitly accessed at a well-known location, rather than implicitly and transiently shared as in LIME. TuCSoN [21] and MARS [4] provide *programmable tuple spaces* supporting event notification and query adaptation through a notion of *reaction*, which is nonetheless rather different from that of LIME. When an agent issues a query on the tuple space, the code associated with a reaction matching the query is executed atomically, albeit asynchronously with respect to the query. While LIME reactions form a core concept for the application programmer, MARS and TuCSoN reactions are meant to be used at the system support level, to provide an intermediate adaptation layer that allows the customization of the way queries are issued and results are obtained for specific classes of agents. Moreover, a tuple space name can be fully qualified with the name of the host where it resides, hence enabling remote operations. Nevertheless, it requires connectivity, and explicit agent knowledge about the tuple space location, as opposed to the LIME model that operates over the current context transparently. Furthermore,

in MARS and TuCSoN mobile agents have access only to the tuple spaces whose location they know, they do not carry tuples as they migrate, and there is no implicit data exchange among tuple spaces.

The KLAIM [20] model supports a programming paradigm where code migrates during execution, using tuple spaces to provide the medium for interaction among processes. Tuple spaces have locality, but unlike in LIME, these tuple spaces are not permanently associated to a process. Instead, KLAIM processes placed at a given locality implicitly interact through the co-located tuple space. There is no transient sharing among tuple spaces, but a process can explicitly interact with any tuple space by identifying its locality, and a process can migrate to a new locality to interact there. While LIME leaves the details of process migration outside the model, KLAIM includes in the formal specification the details of process migration, making it an integral part of the model.

As alluded to in Section 3, the notion of reaction put forth in LIME is profoundly different from similar event notification mechanisms such as those provided by TuCSoN, MARS, TSpaces, and Javaspaces. In these systems, the events respond to *operations* issued by processes on the tuple spaces (e.g., **out**, **rd**, **in**, etc.). In LIME, instead, reactions fire based on the *state* of the tuple space itself. Further, LIME reactions execute as a single atomic step, and cannot be interrupted by other operations. This makes it straightforward for a single LIME reaction to probe for a tuple, react if it is found, and register a reaction if it is not. This same operation in the other systems requires a transaction. Finally, the atomicity of strong reactions increases the power of LIME reactions. For example, with a strong, local reaction, the execution of the listener is guaranteed to fire in the same state in which the matching tuple was found. No such guarantee can be given with an event model where the events are asynchronously delivered. Nonetheless, we support also this second approach through weak reactions.

As work on LIME becomes increasingly recognized, it is being used also as a basis for alternative models. At Purdue University, a group extracted the features of LIME necessary for mobile agents by removing host-level sharing, and created a model referred to as CoreLIME [5]. On top of this restricted model, they proposed some initial ideas for tuple space security. A group at the University of Bologna proposed a calculus-based specification [3] of a model that embeds choices different from the original LIME, including reacting to tuple space operations instead of tuple space contents and blocking agents that generate tuples destined for disconnected agents rather than creating misplaced tuples.

As we conclude this section we note that the effort that went into developing LIME also contributed to the emergence of a more abstract and general coordination concept and methodology called *Global Virtual Data Structures* (GVDS) [27]. It is centered on the notion of constructing individual programs in terms of local actions whose effects can be interpreted at a global level. A LIME group, for instance, can be viewed as consisting of a global set of tuples and a set of agents that act on it in some constrained manner. The set has a structure that changes in accordance with a predefined set of policies and it is this very structure that governs the specific set of tuples accessible to an individual agent through its local interface at any given point in time. The analogy to the concepts of virtual memory and distributed shared memory are very strong and several other research projects have picked up the GVDS theme and instantiated it in their own unique ways. The XMIDDLE [15] system developed at University College of London, for instance, presents the user with a tree data structure based on XML data. When connectivity becomes available, trees belonging to different users can be composed, based on the node tags. Upon disconnection, operations on replicated data are still allowed, and their effect is reconciled when connectivity is restored. Also PEERWARE [7], a project at Politecnico di Milano, exploits a tree data structure, albeit in a rather different way. In PEERWARE, each host is associated with a tree of document containers. When connectivity is available, the trees are shared among hosts, meaning that the document pool available for searching under a given tree node includes the union of the documents at that node on all connected hosts.

7 Conclusions

In this contribution we described LIME, a computational model and middleware specifically designed to support logical mobility of agents and physical mobility of hosts in both wired and wireless settings. LIME reinterprets the notion of tuple space introduced by Linda, and adapts it in an original fashion to the mobile environment. Transparent management of tuple space sharing, contingent on connectivity, offers an effective context awareness mechanism while reactions provide an effective and uniform vehicle for responding to context changes regardless of their nature or trigger. The net result is a simple model with precise semantics and applicability in a wide range of settings, from mobile agent systems operating over wired networks, at one extreme, to mobile ad hoc networks lacking any infrastructure support, at the other. The experience to date with building applications and higher-level middleware layers

with LIME is encouraging, and appears to confirm the value of the conceptual and technological tools put forth by LIME.

Availability. LIME continues to be developed as an open source project, available under GNU's LGPL license. Source code and development notes are available at `lime.sourceforge.net`.

Acknowledgments

This research was supported in part by the National Science Foundation under grant No. CCR-9970939. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the research sponsors.

References

[1] D.E. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[2] G. Blair, N. Davies, A. Friday, and S. Wade. Quality of Service Support in a Mobile Environment: An Approach Based on Tuple Spaces . In *Proc. of the 5^{th} IFIP Int. Wkshp. on Quality of Service (IWQoS'97)*, May 1997.

[3] N. Busi and G. Zavattaro. Some thoughts on transiently shared dataspaces. In *Proc. of the Workshop on Software Engineering and Mobility, co-located with the 23^{rd} Int. Conf. on Software Engineering ICSE)*, 2001.

[4] G. Cabri, L. Leonardi, and F. Zambornelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 2000.

[5] B. Carbunar, M.T. Valente, and J. Vitek. LIME revisited: Reverse engineering an agent communication model. In *International Conference on Mobile Agents*, Atlanta, GA, USA, December 2001.

[6] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus-Linda. In *Workshop on Languages and Models for Coordination, European Conference on Object Oriented Programming*, 1994.

[7] G. Cugola and G.P. Picco. PEERWARE: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, Italy, 2001. Available at `www.elet.polimi.it/upload/picco`.

[8] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[9] D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.

[10] R. Handorean and G.-C. Roman. Service provision in ad hoc networks. In F. Arbab and C. Talcott, editors, *Proceedings of the 5^{th} International Conference on Coordination Models and Languages*, LNCS 2315, pages 207–219. Springer, 2002.

[11] R. Handorean and G.-C. Roman. Secure Sharing of Tuple Spaces in Ad Hoc Settings. In *Proc. of the 1^{st} Int. Workshop on Security Issues in Coordination Models, Languages, and Systems (SecCo 2003)*, Electronic Notes in Theoretical Computer Science (ENTCS), 2003.

[12] IBM. TSpaces Web page. `http://www.almaden.ibm.com/cs/TSpaces`.

[13] JavaSpaces. The JavaSpaces Specification web page. `http://www.sun.com/jini/specs/js-spec.html`.

[14] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.

[15] 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.

[16] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2):97–110, 1998.

[17] Sun Microsystems. Jini web page. `http://www.sun.com/jini`.

[18] 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.

[19] A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Coordination Middleware Supporting Mobility of Hosts and Agents. Technical Report WUCSE-03-21, Dept. of Computer Science, Washington University in St. Louis (MO, USA), May 2003.

[20] 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.

[21] 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.

[22] G.P. Picco. μCODE Web page. `mucode.sourceforge.net`.

[23] G.P. Picco. LIGHTS Web page. `lights.sourceforge.net`.

[24] G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of the 2^{nd} Int. Workshop on Mobile Agents*, LNCS 1477. Springer, 1998.

[25] G.P. Picco and M.L. Buschini. Exploiting transiently shared tuple spaces for location transparent code mobility. In F. Arbab and C. Talcott, editors, *Proc. of the 5^{th} Int. Conf. on Coordination Models and Languages*, LNCS 2315, pages 258–273. Springer, 2002.

[26] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21^{st} Int. Conf. on Software Engineering*, pages 368–377, May 1999.

[27] G.P. Picco, A.L. Murphy, and G.-C. Roman. On global virtual data structures. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, pages 11–29. CRC Press, 2002.

[28] J. Pinakis. *Using Linda as the Basis of an Operating System Microkernel*. PhD thesis, University of Western Australia, Australia, August 1993.

[29] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Proceedings of the 23^{rd} Int. Conf. on Software Engineering*, pages 381–388, Toronto, Canada, May 2001.

[30] 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.

[31] D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6^{th} European Software Engineering Conf. held jointly with the 5^{th} ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, number 1301 in LNCS, Zurich (Switzerland), September 1997. Springer.

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

[33] Lime Team. LIME Web page. `lime.sourceforge.net`.

[34] 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.

[35] S.P. Wade. *An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments*. PhD thesis, Lancaster University, England, September 1999.

[36] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Digest of Papers of the 19^{th} Int. Symp. on Fault-Tolerant Computing*, pages 199–206, June 1989.