Developing Mobile Applications: A LIME Primer

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

August 1, 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 enables significant improvements in the use of communication resources.

Mobility undermines several common assumptions. Disconnection is no longer an infrequent accident: in application 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 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 aimed at supporting mobility at the operating system level (e.g., the work on Coda [13]) or for specific application domains (e.g., repository-based in Bayou [28]). On the other hand, many commercial applications mask mobility by relying on a proxy architecture, but assume the existence of a fixed infrastructure. Clearly, these approaches are limited in that they solve issues that are specific for a given application domain, or do not address unconstrained mobile settings.

LIME (Linda in a Mobile Environment) [21, 16] 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 application

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

programmer with a set of general-purpose programming constructs and, 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, including MANETs.

The design of LIME is inspired by a coordination perspective. 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 in 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 in 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 being able to handle effectively 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 [27], that 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 [16], and the formal semantics of the model [17].[1] The chapter is organized as follows. Section 2 contains an overview of the LIME model. Section 3 walks through a case study application, and shows how its requirements are satisfied by relying on a design exploiting LIME . Section 4 presents instead some higher-level middleware extensions to LIME that we built entirely as an application layer on top of the original middleware. Section 5 places LIME in the context of related work. Finally, Section 6 ends the chapter with some brief concluding remarks.

2 LIME : Linda in a Mobile Environment

The LIME model [21] aims at identifying a coordination layer that can be exploited successfully for designing applications that exhibit logical mobility, physical mobility, or both. The design criteria underlying LIME come from the realization that the problem of designing applications involving mobility can be regarded as a coordination problem [24], and that a fundamental issue to be tackled is the provision of good abstractions for dealing with, and exploiting, a dynamically changing context. To achieve its goal, LIME borrows and adapts the communication model made popular by Linda [9]. After presenting a concise Linda primer, the remainder of this section discusses how the core concepts of Linda are reshaped in the LIME model and embodied in the programming interface of the corresponding middleware implementation.

2.1 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

[1] **GP: Need the TR number**

the previous tuple are all actuals, while the last two fields of ⟨"foo", ?integer, ?float⟩ are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by **in** is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive **rd**(p) operation. Both **in** and **rd** are blocking, i.e., if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple becomes available. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives **inp** and **rdp** , called *probes*, that allow non-blocking access to the tuple space[2]. Moreover, some variants of Linda (e.g., [26]) 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[3].

2.2 The LIME Model

Linda characteristics resonate with the mobile setting. In particular, 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.

2.2.1 The Core Idea: Transparent Context Maintenance

In Linda, the data accessible through the tuple space represents the data *context* available during process interaction. In the model underlying LIME , the shift from a fixed context to a dynamically changing one is accomplished 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 these individual tuple spaces based on connectivity.

The individual tuple space permanently and exclusively attached to a mobile unit is referred to as the *interface tuple space* (ITS) because it provides the only access to the data context for that mobile unit. Each ITS contains the tuples the mobile unit is willing to make available to other units, and access to this data structure uses standard Linda operations, whose semantics remain basically unaffected. These tuples represent the only context accessible to a mobile unit when it is alone.

When multiple mobile units 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 paper, we consider only connectivity. Conceptually, the contents of the ITSs of all group members are merged, or transiently shared, to form a single, large context which 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 LIME , agents may have multiple ITSs distinguished by a name since this is recognized [6] as a useful abstraction to separate related application data. The sharing rule in the case of multiple tuple spaces relies on tuple space names: only identically-named tuple spaces are transiently shared among the members of a group. Thus, for instance, when an agent a owning a single tuple space named X joins a group constituted by an 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.

Transient sharing of the ITS constitutes a very powerful abstraction, as it provides a mobile unit with the illusion of a local tuple space that contains all the tuples coming from all the units belonging to the group, without any need to

[2]Additionally, Linda implementations often include also an **eval** operation which provides dynamic process creation and enables deferred evaluation of tuple fields. For the purposes of this work, however, we do not consider this operation further.

[3]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.

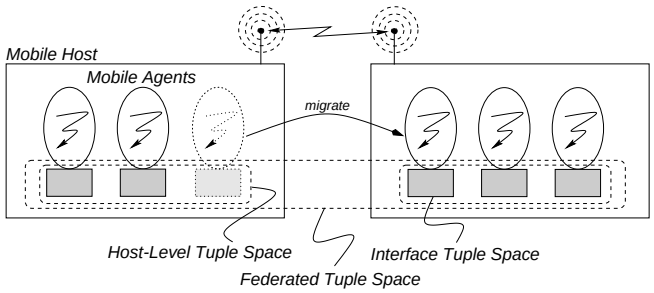

Figure 1: Transiently shared tuple spaces encompass physical and logical mobility.

know the members explicitly. The notion of transiently shared tuple space is a natural adaptation of the Linda tuple space to a mobile environment. When physical mobility is involved, and especially in the radical setting defined by mobile ad hoc networking, there is no stable place to store a persistent tuple space. Connections among machines come and go and the tuple space must be partitioned in some way. Analogously, in the scenario of logical mobility, maintaining locality of tuples with respect to the agent they belong to may be complicated. LIME enforces an a priori partitioning of the tuple space in subspaces that get transiently shared according to precise rules, providing a tuple space abstraction that depends on connectivity.

2.2.2 Encompassing Physical and Logical Mobility

In LIME , mobile hosts are connected when a communication link is available. Availability may depend on a variety of factors, including quality of service, security considerations, or connection cost; all of which can be represented in LIME , although in this paper we limit ourselves to availability determined by the presence of a functioning link. 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 a special case of connection and disconnection, respectively. Figure 1 depicts the model adopted by LIME . Mobile agents are the only active components; mobile hosts are mainly roaming containers which provide connectivity and execution support for agents. In other words, mobile agents are the only components that carry a "concrete" tuple space with them.

The transiently shared ITSs belonging to multiple agents co-located on a host define a *host-level tuple space*. The concept of transient sharing can also be applied to the host-level tuple spaces of connected hosts, forming a *federated tuple space*. When a federated tuple space is established, a query on the 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.

In this model, physical and logical mobility are separated in two different tiers of abstraction. Nevertheless, many applications do not need both forms of mobility, and 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. 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.

Nevertheless, it is interesting to note how mobility is not dealt with directly in LIME , i.e., there are no constructs for triggering the mobility of agents or hosts. Instead, the effect of migration is made indirectly manifest to the model and middleware only through the changes observed in the connectivity among components. This choice, that sets the nature of mobility aside, keeps our model as general as possible and, at the same time, enables different instantiations of the model based on different notions of connectivity.

2.2.3 Controlling Context Awareness

Thus far, LIME appears to 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 has the potential for greatly

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.

simplifying 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 domains where the designer needs more fine-grained control over the portion of the context that needs to 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, 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 $\mathbf{out}$ 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 $\mathbf{out}(t)$, the tuple t is inserted in the ITS of the agent calling the operation, say ω. At this point the tuple t has a current location ω, and a destination location λ. If the agent λ is currently connected, the tuple t is moved to the destination location in the same atomic step. 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[4], will remain misplaced unless λ becomes connected. In the latter 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 $\mathbf{in}$ and $\mathbf{rd}$ operations that allow 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.

2.2.4 Reacting to Changes in Context

In the fluid scenario we target, the set of available data, hosts, and agents change rapidly according to the reconfiguration induced by mobility. Reacting to changes constitutes a significant fraction of an application's activities. At first glance, the Linda model would seem sufficient to provide some degree of reactivity by representing relevant events as tuples, and by using the $\mathbf{in}$ operation to execute the corresponding reaction as soon as the event tuple appears in the tuple space. Nevertheless, in practice this solution has a number of drawbacks. For instance, programming becomes cumbersome, since the burden of implementing a reactive behavior is placed on the programmer rather than the system. Moreover, enabling an asynchronous reaction would require the execution of $\mathbf{in}$ in a separate thread of control, hence degrading performance. Therefore, LIME explicitly extends the basic Linda tuple space with 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. The semantics of reactions are based on the Mobile UNITY

[4]Note how specifying a destination location λ implies neither guaranteed delivery nor ownership of the tuple t to λ. Linda rules for nondeterministic 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 .

reactive statements [15], described formally in a later section. Informally, 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.

LIME reactions can be explicitly registered and deregistered on a tuple space, and hence do not necessarily exist throughout the life of the system. Moreover, a notion of *mode* 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, and hence removed from the reactive program. Instead, a reaction registered with mode ONCEPERTUPLE is allowed to fire an arbitrary number of times, but never twice for the same tuple. Finally, reactions can be annotated with location parameters, with the same meaning discussed earlier for **in** and **rd** . Hence, the full form of a LIME reaction is $\mathcal{R}[\omega, \lambda](s, p, m)$, where m is the mode.

Reactions provide the programmer with very powerful constructs. They enable the specification of the appropriate actions that need to take place in response to a *state* change and allow 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 [25], including those exploited by tuple space middleware such as TSpaces [11] and JavaSpaces [12], that are typically stateless and provide no guarantee about the atomicity of event reactions.

Nevertheless, this expressive power comes at a price. In particular, when multiple hosts are present, the content of the federated tuple space depends on the content of the tuple spaces belonging to physically distributed, remote agents. Thus, maintaining the requirements of atomicity and serialization imposed by reactive statements requires a distributed transaction encompassing several hosts for every tuple space operation on any ITS —very often, an impractical solution. For specific applications and scenarios, e.g., those involving a very limited number of nodes, these kind of reactions, referred to as *strong reactions*, would still be reasonable and therefore they remain part of the model. 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 introduce a new 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.

2.2.5 Exposing System Configuration

It is interesting to note that the extension of Linda operations with location parameters, as well as the other operations discussed thus far, foster a model that 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 in the portion of context concerned with application data. Although this perspective is often enough for many mobile applications, in many others the portion of context more closely related to the system configuration 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 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 tuple spaces they are sharing or, for mobile agents, which host they reside on. 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

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

changes in the configuration of the system.

Together, the LimeSystem tuple space and the other application-defined transiently shared tuple spaces enable the definition of a fully context aware style of computing.

2.3 Programming with LIME

We complete the presentation of the LIME model by concisely illustrating the application programming interface provided in the current implementation[5] of LIME.

The class `LimeTupleSpace`, whose public interface is shown[6] in Figure 2, embodies the concept of a transiently shared tuple space. In the current implementation, agents are single-threaded and only the thread of the agent which 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. The name of the tuple space is specified as a parameter of the constructor.

Agents may also have *private* tuple spaces, i.e., not subject to sharing and not appearing in the LimeSystem tuple space. A private `LimeTupleSpace` 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, or it can be useful as a primitive data structure for local data storage. All tuple spaces are initially created private, and sharing must be explicitly enabled by calling the instance method `setShared`. 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 properties 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 2). 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 towards the run-time support, and 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 define whether transient sharing is constrained to a host-level tuple space, or whether it spans the whole federated tuple space.

`LimeTupleSpace` contains the Linda operations needed to access the tuple space, as well as the operation variants annotated with location parameters. The only requirement for tuple objects is to implement the interface `ITuple`, which is defined in a separate package providing access to a lightweight tuple space implementation. As for location parameters, LIME provides two classes, `AgentLocation` and `HostLocation`, which extend the common superclass `Location`, enabling the definition of globally unique location identifiers for hosts and agents. Objects of these classes are used to specify different scopes for LIME operations, as described earlier. 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

[5] The LIME Web site [27] contains extensive documentation and programming examples.

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

```
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.

refer the whole host-level tuple space, if `cur` is of type `HostLocation`. The constant `Location.UNSPECIFIED` is used to allow any location parameter to match. Thus, 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 location 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 "concrete" tuple spaces in LIME. In the current implementation of LIME, 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.

All the operations retain the same semantics on a private tuple space as on a shared tuple space, except for blocking operations. Since the private tuple space is exclusively associated to one agent, the execution of a blocking operation when no matching tuple is present would suspend the agent forever, effectively waiting for a tuple that no other agent can possibly insert. Hence, blocking operations always generate a run-time exception when invoked on a private tuple space.

The remainder of the interface of `LimeTupleSpace` is devoted to managing reactions; other relevant classes for this task are shown in Figure 3. Reactions can either be 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. The type of a reaction is used to enforce the proper constraints on the registration through type checking. These two classes share the abstract class `Reaction` as a common ancestor, which 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 mode. 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. The reaction mode can be either of the constants `ONCE` or `ONCEPERTUPLE`, defined in `Reaction`. Reactions are added to the ITS by calling either `addStrongReaction` or `addWeakReaction`, depending on the desired semantics. As we discussed earlier, in the current implementation strong reactions are confined to a single host, and hence only a `LocalizedReaction` can be passed to the first method. Registration of a reaction returns an object `RegisteredReaction`, that 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.

3 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 .

3.1 Requirements

The goal is to design a jigsaw assembly game for multiple players in the ad hoc mobile 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 where 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 quickly updated 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 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. Additionally, a single player should be able to participate in more than one puzzle at a time.

3.2 Design

The requirements sketched above are intentionally vague, leaving many options open to the designer. Here we sketch the design developed by an undergraduate student as a course project. The design takes into consideration the programming style encouraged by LIME as well as the constraints of the wireless, mobile environment.

One of the first choices for all LIME applications is the use of the federated tuple space(s) and the format of the tuples. In order to support multiple concurrent tuple spaces, an obvious choice is to use a separate tuple space for each concurrent puzzle. This effectively separates the actions involving the distinct puzzles, and easily allows a player to participate in as many puzzles as they would like. In order to find active puzzles, a player need only look into the LimeSystem to find the tuple spaces names. By creating a tuple space with one of these names, the player joins the game.

Each tuple space is used as the repository for the current status information about a single puzzle, including the pieces as well as how they are assembled. A critical choice is how to use tuples to represent this information. We represent the puzzle with tuples with two distinct patterns: *bitmap* and *assembly*. Bitmap tuples represent the individual puzzle pieces and contain two fields: the puzzle piece identifier and the bitmap of the piece. The second tuple pattern, the assembly, represents a group of connected puzzle pieces. The tuple itself contains a single field with the list of the identifiers of the connected pieces. When a new game starts, two tuples are inserted for each puzzle piece, one containing the bitmap and one containing a list with a single entry, the piece identifier. When two pieces are joined, the two assembly tuples of the original pieces are replaced by a single tuple representing the change.

Two observations can be made about our choice of representing the puzzle in this manner. First, when puzzle pieces are assembled, the bitmap tuples do not change. Because these tuples contain image information they are likely to be large in comparison to the assembly tuples. By requiring only the assembly tuples to change, we save computation time to remove and reinsert the large image information each time an assembly is made. Similarly, we save in bandwidth usage as the bitmap images are not as frequently transmitted over the wireless link. This will become more apparent when we discuss the reactions.

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 are present. However, when the players are disconnected, the puzzle pieces are dividend among the tuple spaces of the players, and therefore are not accessible to everyone. We leave the choice of which tuples go with which players up to the players themselves. Specifically, when play begins, the player that initiated the game is the "owner" of all of the pieces, both bitmaps and assemblies. The puzzle game contains a selection mechanism that allows a player to choose a piece out of the federated tuple space and change the LIME "current" location to match itself. This means that when the player disconnects, the pieces it has selected remain accessible. This notion of ownership is on the assembly level, but must also be applied to bitmaps. In other words, when a player takes ownership of an assembly, it must also take ownership of the bitmaps associated with the pieces of the assembly.

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

3.3 Implementation

The implementation of the puzzle is remarkably simple considering the complexity of the resulting application. In fact, the code is disproportionally devoted to the graphics, with very little needed for LIME . The application is essentially event driven, responding to interface requests by the user and remote operations observed through registered reactions.

The main display for each player is one *puzzle tray* for each puzzle they are participating in. For simplicity, we consider only a single puzzle. Initially, the tray is empty, but it is populated when the player starts the puzzle. This corresponds to the population of the tuple space with the bitmap and assembly tuples. In LIME , this is accomplished with a sequence of **out** operations.

The updating of the puzzle tray is actually accomplished by a ONCEPERTUPLE LIME weak reaction on the federated tuple space registered for assembly tuples. When this reaction fires, if the player has not already displayed the bitmap of the puzzle pieces identified in the assembly tuple's list, the reaction issues an **rdp** operation to retrieve the bitmap tuple. Subsequently, the puzzle tray is updated with the new puzzle piece. One detail worth mentioning is that the **rdp** operation must specify the current location tuple space from which to retrieve the bitmap tuple. Since the assembly tuple and the bitmap tuple reside in the same space, and the `ReactionEvent` contains the source of the assembly tuple, we use the same information to retrieve the bitmap.[7]

When a piece is selected for ownership by a player, the result is the movement of the tuples representing the piece from the old player to the new player. [8] In LIME , this is accomplished by performing two **inp** operations to retrieve both the bitmap and assembly tuples, followed by two **out** operations to reinsert the tuples into the tuple space. Because we do not specify a destination location for the **out** operations, the default is that the tuples remain with the current field set to the new player. Again, the **inp** operation must specify the current location tuple space from which to retrieve the tuples, but this information is easily kept with the assembly tuple once it has been retrieved through the reaction. To make it apparent to 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. This can be seen in Figure 4.

Interestingly, when a player selects a piece, the **out** operation that reinserts the assembly tuple causes the earlier described reaction to fire. This time, because the graphic for the tuple has already been displayed, the bitmap tuple is not retrieved, but the screen is updated to reflect the change in the outline color of the puzzle piece.

Assembly of two pieces is quite 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. As before, the reaction fires and the puzzle trays of all connected players are updated. There is one subtle point worth mentioning:[9] what happens if one of the **inp** operations returns `null`? For example, if player p_1 is trying to assemble pieces t_1 and t_2, and at the same time, player p_2 is trying to select piece t_2 from p_1's tuple space. It may

[7] **GCR: I found this paragraph very confusing. The use of the term current location to explain things is confusing even for me. Refer to agents and their tuple spaces and introduce the "location" as a detailed technical explanation after the ideas are made clear.**

[8] **GCR: If the operations are always in order, the possibliity of deadlock is removed.**

[9] **GCR: You are using this too much in teh entire section. Rephrase.**

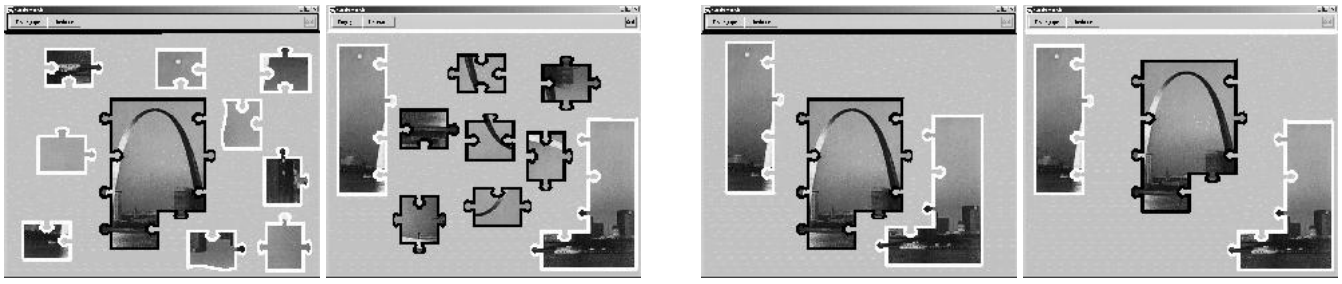

Figure 4: ROAMINGJIGSAW. The left two images show the puzzle trays of the black and white players while they are disconnected and able to assemble only their selected pieces. The right two images show the black and white puzzle trays after the players re-engage and see the assemblies that occurred during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

happen that p_1 successfully issues the first **inp** , removing t_1. Second, p_2's **inp** to remove t_2 successfully executes, and p_2 becomes the owner of the tuple. Finally, p_1's second **inp** issues, but returns `null`. In this case, p_1 does not have sufficient data to complete the assembly, and must reinsert t_1 into the tuple space. To the user, we indicate this failure with an audible beep.

When a player disconnects, there is no immediate change in the puzzle tray to indicate the disconnection. Therefore, it is possible for a player to try to select for ownership a piece visible in the tray. In this case, the **inp** will return `null`, and we report the failure as before.

When two players reconnect after a disconnection, the requirements outlined previously state that the puzzle trays of the players must be updated to reflect the changes made during the disconnection. Because these changes are represented by assembly tuples, the reaction described earlier will fire on these tuples, and the display will be updated. Figure 4 shows the appearance of the puzzle tray during disconnection and after reconnection. No additional mechanisms are needed to explicitly handle reconnection.

It is worth emphasizing[10] that the entire operation of the puzzle relies on this single reaction to assembly tuples and exploits only three LIME operations (**inp** , **rdp** , and **out**).

3.4 Beyond the Puzzle

From the description, it is evident that ROAMINGJIGSAW 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. and to tolerate temporary disconnections.

ROAMINGJIGSAW 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 in ROAMINGJIGSAW may be adapted easily to handle updates in the data being shared by real applications. One example could be provided by collaborative work applications involving mobile users, where our mechanism could be used to deal with changes in sections of a document, or with paper submissions and reviews to be evaluated by a program committee.

4 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 , as described in the previous section, gave us the opportunity to evaluate the expressive power of LIME constructs in building mobile applications. In this section, instead, we present a couple of 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.

[10] **GCR: Sounds repetitious.**

4.1 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 [20] 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 potentially for fundamentally changing 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 [8] approach 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 nodes.

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 [20], can operate on the LIME tuple space associated to 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 of 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 needs 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 even 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 even to 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 add even more degrees of freedom, by allowing agents to monitor the federated code base and react to changes with different atomicity guarantees. Reactions can be exploited straightforwardly 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.

4.2 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 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 includes 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 ad hoc settings with minimal programming effort. Later efforts built upon this results 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 and tuple spaces in general and federated registries in particular.

5 Related Work

The last several years have seen a revitalization of Linda for distributed computing applications, including mobile environments. From the industrial perspective, both Sun and IBM have developed tuple space implementations for client-server coordination, i.e., JavaSpaces [12] and TSpaces [11], respectively. These systems present a centralized tuple space, accessible through remote operations by multiple processes. Their client-server approach is significantly different from that of LIME , as we target mobile ad hoc applications with implicit access to the remote data of other hosts, and support mobile agents.

Distributed Linda implementations have been studied extensively for fault tolerance [30, 1] and data availability [23]. The main disadvantage with these approaches is their need for high degrees of connectivity among the hosts of the distributed portions of the tuple spaces, a property inherently not present in the mobile environment.

One of the first applications of Linda to mobility came in the Limbo platform [2, 29], a system that builds the notion of quality of service aware tuple spaces that reside on mobile hosts. The quality of service information itself is stored in the tuple spaces and can be made accessible to agents on remote hosts. While Limbo has a notion of distributed tuple spaces that span multiple hosts, there are no mobile agents carrying tuple spaces when the migrate, no concept of reaction, and the mechanism for relocation of tuples is unclear. Interestingly, the Limbo *universal tuple spaces*, which serves as a registry for all tuple spaces is similar to the LimeSystem tuple space of LIME . However, instead of describing the *current* system context, the universal tuple space remembers all tuple spaces the host has ever encountered without regard for current reachability.

Two other models, TuCSoN [19] and MARS [4], exploit tuple space coordination for mobile agents, creating *programmable tuple spaces*. When an agent poses a query to the tuple space, the registered reaction that matches the operation fires, and an action is atomically performed. While in LIME reactions form a core concept for the application programmer, MARS and TuCSoN reactions are designed to be implemented by the system support designer to provide an intermediate access between the form of the query (which can vary among agents) and the data (which remains constant within a host, but is adapted when a query arrives). Another feature of MARS and TuCSoN is the option to fully qualify a tuple space name, identifying the specific host where the tuple space resides. This enables remote operation on tuple spaces, but connectivity must be available and the agent must be explicit about interaction, as opposed to the LIME model that operates over the current context transparently. Further, in MARS and TuCSoN, mobile agents only have access to the tuple spaces fixed at the hosts, they do not carry tuples as they migrate, and there is no coordination or data exchange among tuple spaces.

The KLAIM [18] 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 located 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 locally. 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 the informal description of LIME provided in Section 2, the notion of reaction put forth in LIME is profoundly different from similar event notification mechanisms such as those provided by TuCSoN, 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 , however, 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 this second approach through weak reactions. [11]

The work on LIME has also been used 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, it should be noted 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) [22]. 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 [14] 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. [12]

6 Conclusions

LIME is a middleware specifically designed to support logical mobility of agents and physical mobility of hosts in both wired and wireless settings. Within this general context, its distinctive feature is the reliance on coordination to simplify the development of mobile applications. While building on the decoupling advantages of the original Linda model, LIME breaks new ground by extending coordination technology to mobile systems, including the ad hoc wireless setting. Transparent management of tuple space sharing, contingent of 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 ad hoc networks lacking any infrastructure support, at the other. While a full formal validation of LIME 's impact on software development productivity is still to be performed, our experience to date with the development of a reasonable set of applications in wireless settings appears to validate LIME 's potential for rapid development of mobile applications. Finally, it should be noted that LIME , in addition to demonstrating the practical use of coordination technology in mobile computing, opens a new area of research involving the application of state-based coordination models and middleware to context-aware computing.

[11] **GP: Up to here is the content of Section 5.3 of the formal Lime paper, unchanged...**

[12] **GP: This is section 5.2 with minor modifications.**

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] 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.