LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents

AMY L. MURPHY
University of Lugano, Switzerland
GIAN PIETRO PICCO
Politecnico di Milano, Italy
and
GRUIA-CATALIN ROMAN
Washington University in St. Louis, MO, USA

LIME (Linda in a Mobile Environment) is a model and middleware supporting the development of applications that exhibit physical mobility of hosts, logical mobility of agents, or both. LIME adopts a coordination perspective inspired by work on the Linda model. The context for computation, represented in Linda by a globally accessible, persistent tuple space, is refined in LIME to transient sharing of identically-named tuple spaces carried by individual mobile units. Tuple spaces are also extended with a notion of location and programs are given the ability to react to specified states. The resulting model provides a minimalist set of abstractions that facilitate rapid and dependable development of mobile applications. In this paper, we illustrate the model underlying LIME, provide a formal semantic characterization for the operations it makes available to the application developer, present its current design and implementation, and discuss lessons learned in developing applications that involve physical mobility.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Algorithms, Design

Additional Key Words and Phrases: Mobile Computing, Tuple Spaces, Middleware

1. INTRODUCTION

In the arena of modern distributed computing, mobility is emerging as a disruptive new trend that challenges fundamental assumptions across the board, from theoretical foundations to software engineering practices. Powerful social forces energized by advances in wireless communication, device miniaturization, and new software design techniques are creating a growing demand for applications that exploit and support *physical mobility* of hosts moving through space while maintaining connec-

Author addresses: A.L. Murphy amy.murphy@unisi.ch, G.P. Picco picco@elet.polimi.it, G.-C. Roman roman@cs.wustl.edu.

tions with other hosts. At the same time, *logical mobility* has emerged as a novel architectural style that removes the static binding between software components and hosts, and enables run-time component migration for improved flexibility and performance. These two forms of mobility complement each other, meaning logical mobility can provide the fluid software fabric necessary to cope with the high dynamicity imposed by physical mobility. Rapid development of mobile applications demands a new way of thinking and aggressive experimentation if a new set of best design practices is to emerge soon. The basic premise of this paper is that coordination technology can be extended for use in mobile computing and can offer an elegant solution to a set of difficult engineering problems.

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 [Gelernter 1985], this can be achieved by allowing independently developed agents 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 agents 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. The transition to mobility requires one to revisit the basic model with a new intellectual bias. The process entails accommodating physical and logical distribution of tuples and the movement of hosts and agents through physical or logical spaces.

LIME (Linda In a Mobile Environment) is our response to the software engineering challenge posed by the advent of mobility. It defines a novel coordination-based approach to the development of mobile applications. When it appeared [Picco et al. 1999], LIME was the first coordination model and middleware to address the need to integrate concerns having to do with physical mobility of hosts and logical mobility of agents. The LIME computational model assumes a set of hosts that act as containers in which agents are located. Physical connectivity among the hosts is supported by wired or wireless links and may be altered by mobility or by explicit connection and disconnection. Agents can move from one host to another reachable host of their own volition. LIME preserves the essence of the Linda model, its simplicity and decoupled style of computing, by continuing to channel all coordination actions through a simple interface perceived by each agent to be merely a local tuple space. Access to the tuple space is carried out using an extended set of tuple space operations that includes several novel constructs designed to facilitate flexible and timely responses to changes in the contents of the tuple space. Each agent may own multiple tuple spaces that may be shared with other agents within communication range. Sharing is made manifest by logically extending the contents of each tuple space to include the tuples present in all participating tuple spaces. The set of tuples being shared changes over time as a result of the agents' local control regarding sharing and in response to the mobility

of both agents and hosts. When hosts come into communication range the set of shared tuple spaces expands and when they move apart it contracts. The net result is a transparently managed context that expresses itself in terms of changes in the contents of what otherwise appear to be local tuple spaces. The agent behavior is altered both by the availability of new data and by its reactive responses to contextual changes.

The development process of LIME entailed a close interplay between formal semantic definition, implementation pragmatics, and application-driven evaluation of the resulting model and middleware. The insistence on formalizing the model and the semantics of the API is rooted in the conviction that precise semantics are key to dependable development, particularly when working in a novel and demanding setting such as mobility. Implementation considerations led to weakening of certain constructs, to the introduction of features the formal framework did not identify, and to the enhancement of the LIME middleware so as to ensure its applicability in a wide range of physical and logical mobile settings. Overall, the focus on application development and continuous empirical evaluation contributed to practically minded additions to the model. Ultimately, the effort culminated in a Java-based implementation of the LIME middleware [Murphy et al. 2001], currently available[1] as an evolving, open source project. Several application development exercises with programmers possessing varying skill levels reinforced our conviction that a properly tailored model of coordination, such as LIME, can be an effective software engineering tool in the mobile setting.

This paper constitutes a complete description of LIME, from the model to the middleware and applications. The semantics of the LIME model is first described informally in Section 2, and then formalized in Section 3. Based on these grounds, Section 4 describes the LIME middleware embodying the model concepts. The application programming interface is presented, together with an overview of the middleware architecture. Section 5 reviews our experience with several applications developed using LIME. Finally, Section 6 discusses lessons learned and related work, and Section 7 ends the paper with some brief concluding remarks.

2. THE LIME MODEL

The LIME model [Picco et al. 1999] 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 perspective that the problem of designing applications involving mobility can be regarded as a coordination problem [Roman et al. 2000], 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 coordination model made popular by Linda [Gelernter 1985]. After presenting a concise Linda primer, the rest of this section discusses how the core concepts of Linda are reshaped in the LIME model. The presentation is kept informal; the model is formalized in Section 3. Also, in this section we do not discuss implementation issues. Instead, they are addressed in Section 4.

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

2.1 Linda in a Nutshell

In Linda, processes coordinate 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 **out**(t) operation, and can be removed by executing **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 **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., [Rowstron 1998]) 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 follows that of probes.

2.2 LIME: Linda in a Mobile Environment

Linda characteristics resonate with the mobile setting. In particular, coordination among processes in Linda is decoupled in *time* and *space*, i.e., producers and consumers 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 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 coordinating 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. This is especially true in a mobile ad hoc environment, where components move in and out of range and there is no single location for a tuple space repository to be placed such that it will always remain accessible to all components.

Our approach, as explained in the following, moves Linda into the mobile environment while retaining the core of the original Linda philosophy.

[2]Linda defines also an **eval** operation that 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.

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 agent, 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 agent, is referred to as the *interface tuple space* (ITS) because it provides the only access to the data context for that mobile agent. Each ITS contains the tuples the mobile agent is willing to make available to other agents, 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 agent when it is alone.

When multiple mobile agents are able to communicate, either directly or transitively, we say these agents form a LIME *group*. For example, two agents on the same host can form a group. If two hosts can communicate with one another, all the agents on those hosts form a group. Although the notion of group can be based on more than just communication, 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 that is accessed by each agent through its own ITS. The sharing itself is transparent to each mobile agent, 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 agent, 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 agent 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 [Carriero et al. 1995] 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 including 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 agent with the illusion of a local tuple space that contains all the tuples coming from all the agents belonging to the group, without any need to 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

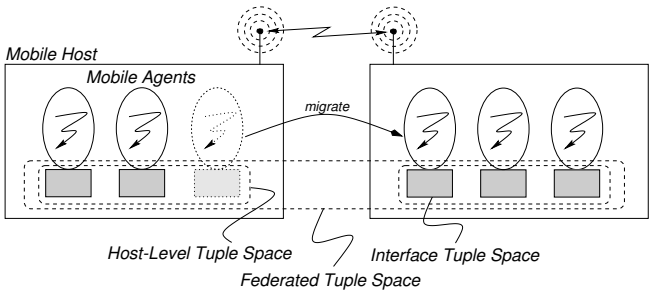

Fig. 1. Transiently shared tuple spaces encompass physical and logical mobility.

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 are 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; however 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 that provide connectivity and execution support for agents. In other words, mobile agents are the only components that carry a "concrete" tuple space.

The transiently shared ITSs belonging to multiple agents co-located on a host define a *host-level tuple space*. The concept of transient sharing is also 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 form 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.

It is interesting to note that mobility is not dealt with directly in LIME, i.e., there are no constructs for instructing either a host or agent to move to a new location. Instead, the *effect* of mobility is indirectly manifest only in the changes observed in the tuples forming the host-level and federated tuple spaces. This choice, that sets the nature of mobility aside, keeps our model as general as possible.

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 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 agent 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 **out**$[\lambda]$ operation extends **out** with a location parameter representing the identifier of the agent responsible for holding the tuple. The semantics of **out**$[\lambda](t)$ involve two steps. The first step is equivalent to a conventional **out**(t), the tuple t is inserted in the ITS of the agent[3] 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, identified as such because its current and destination values are not equal, if not withdrawn[4], remains 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, changing its current location to λ. By using **out**$[\lambda]$, the caller can specify that the tuple is

[3]For notational convenience, **out**$[\lambda](t)$ and **out**(t) are equivalent when the agent issuing the operation is λ.

[4]Note how 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, some other agent may withdraw t from the tuple space before λ, even after t reached λ's ITS.

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 agent 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 agent ω's tuple space and destined to λ
Ω	λ	Tuples in Ω's host-level tuple space and destined to λ

Table I. 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.

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 that exploit 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[5] in Table I. The current location parameter enables the restriction of scope from the entire federated tuple space (no value specified) to the tuple space of all agents on a given host, Ω, or even a given agent, ω. 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 frequently 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 **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 reactive behavior is placed on the programmer rather than the system. Moreover, enabling an asynchronous reaction would require the execution of **in** in a separate thread of control, hence adding overhead.

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 reactive statements [McCann and Roman 1998], described formally in a later section. Informally, a reaction can *fire* if a tuple matching pattern p exists in the tuple space. After one of these regular tuple space operations, 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

[5]The non-annotated version of $\mathbf{in}(p)$ and $\mathbf{rd}(p)$ are equivalent to the annotated versions with current and destination unspecified.

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 keeping 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 [Rosenblum and Wolf 1997], including those exploited by tuple space middleware such as TSpaces [Lehman et al. 2001] and JavaSpaces [Freeman et al. 1999], which 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 a form of reactivity that spans the whole federated tuple space but exhibits 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 the departure of a mobile agent, or to determine the set of agents currently belonging to a LIME group. 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 agents present in the group and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, on which host they reside. 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.

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.

3. A FORMAL SEMANTICS FOR LIME

The ultimate goal of our research is rapid development of dependable mobile applications. We contend that a precise understanding of the underlying model and its implementation is essential to achieving a high level of dependability. Of course, highly complex formal specifications may be ignored or may actually become a source of confusion for the middleware user. For this reason we opted in favor of modeling the semantics of LIME by using an existing operational model called Mobile UNITY [McCann and Roman 1998; Roman et al. 1997], a state-based model consisting of a notation system and associated proof logic, which extends the original UNITY [Chandy and Misra 1988] model. Mobile UNITY was specifically designed to express mobility in all its forms and to enable one to reason formally about systems of mobile components, and it has been successfully applied to this end [Picco et al. 2001; McCann and Roman 1999].

The formal definition of the semantics of LIME was a fundamental constituent of its development, done in close connection with the actual implementation of the middleware. Formalization and implementation continuously contributed one to the other. The former inspired novel programming constructs (e.g., the notions of transient sharing and reaction borrowed from Mobile UNITY) with well-founded semantics; the latter checked the feasibility of these constructs against the reality of development, uncovering pragmatic needs (e.g., the need for weak reactions). In a sense, the LIME middleware has been shaped by the formal semantics of the LIME

model—and vice versa.

The formal semantics we defined, however, is also valuable per se. First, by providing a precise specification of the middleware behavior, it helps understanding even the smaller semantic details without having to resort to the source code. As such, it can be exploited as a common language and a stepping stone for similar research efforts. Indeed, in our group we used it as such for some of the evolutions of LIME. Second, the Mobile UNITY proof logic, in combination with the LIME specification and an application specification allow one to reason formally about the behavior of a complex, mobile application.

The remainder of this section is divided in two parts. In Section 3.1 we present a formal specification of the Linda model which LIME builds upon. Section 3.2 defines the semantics of LIME. Despite reliance on Mobile UNITY, the presentation is self-contained and does not require any prior knowledge of this formal language.

3.1 Formalizing Linda

This section progressively introduces the formal semantics for Linda. It begins with a high level view and an example, then drops down to a brief description of the syntax and semantics of Mobile UNITY, and finally describes formally the Linda data structures and operations.

Figure 2 shows a Mobile UNITY specification[6] for a Linda-based producer-consumer system where jobs are exchanged through the tuple space. The producer randomly and continuously generates jobs of different names (A and B), putting the job name (randomly determined at initialization time) and its description into a tuple space called *jobs*. Each of the two consumers removes jobs one at a time from the tuple space, and performs the corresponding actions.

To understand the Linda components of the example, it is first necessary to understand the structure given by Mobile UNITY. The **Program** sections describe the behavior of each kind of concurrent program[7]. The **Components** section defines the program instances (or agents) that make up the system. Instances of the same program are distinguished by an identifier that is set at creation time, and the statements of the program are essentially duplicated for each component instance. Each program contains a **declare** section for naming variables and declaring their types, and an **initially** section for defining the allowed initial values of the variables. Variables that are not explicitly assigned an initial value in the **initially** section assume an arbitrary value compatible with the type. The **assign** section specifies the guarded assignments that define the state transitions of the system. System execution involves non-deterministically selecting a statement in a weakly fair manner from one of the component programs, and evaluating its guard. If it is true, the statement is executed, otherwise it is skipped. Statements are separated by the $[]$ operator. The $\|$ operator is used to construct multiple assignments to be executed in a single atomic step. For example, the consumer program contains two

[6]In the following, we use different fonts for improving the readability of our semantics, by distinguishing among constructs that are available to the specifier as part of LIME or Mobile UNITY (e.g., **in** and **reacts-to**), auxiliary macros used for structuring the specification but not available to the programmer (e.g., copy), types (e.g., Tuple), constants and labels (e.g., CUR), and normal variable names (e.g., *events*).

[7]For now we ignore the λ variable on each program.

System *ProducerConsumer*
  Program *Producer(i)* **at** λ
    declare
      $jobs$: TupleSpace
      $jobName$: $\{\varepsilon, \text{A}, \text{B}\}$
      $jobInfo$: String
    initially
      $jobs = \{\}$ ▯ $jobName = \varepsilon$
    assign
      $jobName := \text{A}$ ‖ $jobInfo :=$ newInfo(A)
      ▯ $jobName := \text{B}$ ‖ $jobInfo :=$ newInfo(B)
      ▯ (**out**($jobs$, **createTuple**($jobName$, $jobInfo$)) ‖ $jobName = \varepsilon$)
                                                if $jobName \neq \varepsilon$
  end
  Program *Consumer(i)* **at** λ
    declare
      $jobs$: TupleSpace
      $curJob$: Tuple
      $jobName$: $\{\text{A}, \text{B}\}$
    initially
      $jobs = \emptyset$ ▯ $curJob = \varepsilon$
    assign
      $curJob :=$ **in**($jobs$, **createTuple**($jobName$, String)) **if** $curJob = \varepsilon$
      ▯ $curJob := \varepsilon$ ‖ doJob($curJob$) **if** $curJob \neq \varepsilon$
  end
  Components
      Producer(0) ▯ *Consumer(1)* ▯ *Consumer(2)*
  Interactions
      $\langle \forall\, a, b, T : a, b \in \mathcal{C}, T \in \mathcal{T} :: a.T \approx b.T \rangle$
end

Fig. 2. A Linda producer-consumer specified in Mobile UNITY.

statements; the first removes a job tuple from the tuple space, while the second simulates the performance of a job through a function doJob, whose details are not relevant here, and resets the value of *curJob* to ε in the same atomic step. The guard in the first statement prevents the consumer from taking a new job before the previous one is completed (i.e., before *curJob* is reset to ε).

In Mobile UNITY, different from the original UNITY, variables with the same name in different programs are distinct. For example, the tuple space variable *jobs* in the producer is distinct from the one in either of the consumers. Nevertheless, in Linda these tuple spaces should always be the same. To accomplish this in Mobile UNITY, we exploit the sharing construct $\approx$, which allows one to express symmetric and transitive sharing between variables belonging to different programs. Let us assume that $\mathcal{C}$ is the set of all component names and $\mathcal{T}$ is the set of names of all the variables of type TupleSpace. In our example, $\mathcal{C} = \{$*Producer(0)*, *Consumer(1)*, *Consumer(2)*$\}$, and $\mathcal{T} = \{jobs\}$.

Hence, by stating[8] in the **Interactions** section that:

$$\langle \forall\, a, b, T : a, b \in \mathcal{C}, T \in \mathcal{T} :: a.T \approx b.T \rangle$$

we force any tuple spaces with the same variable name to be shared, even if they are declared within the name spaces of different agents. In our example, the *jobs* tuple spaces of the producer and consumers become shared by virtue of this statement; hence, when the producer writes a tuple to its tuple space it is immediately available in the consumers' tuple space. Moreover, in this case sharing is unconditional, that is, the tuple spaces are *always* shared. When describing LIME, we remove this assumption and specify the conditions under which transient sharing is enabled, by exploiting more sophisticated forms of the sharing construct.

Placement of the above statement in the **Interactions** section is intentional. This section of the specification is reserved in Mobile UNITY for statements that involve more than one component. The formal semantics of Linda, and next of LIME, is hence constituted by two parts. The first one is the statements we place in the **Interactions** section, which effectively capture a significant part of the run-time communication among programs, as enabled by the middleware. The **Interactions** section of *every* specification exploiting the LIME model must contain the **Interactions** statements we describe in the LIME formal semantics. The second part is the definition of the various constructs, such as the Linda operations **in** and **out** appearing in the example, that are available to the programmer. These constructs are technically defined as *macros*, whose meaning is represented in terms of the basic Mobile UNITY statements. The examples we use here for Linda and next for LIME provide the reader with the opportunity to see how these two constituents of our formal semantics are exploited in the specification of an application.

In this section, we begin the formal description of Linda by focusing first on the essential data structures, then on the operations.

Data Structures. The fundamental data structures used in Linda are tuples and tuple spaces. Tuples are represented by the type Tuple, instances of which are generated by using the **createTuple** function. Parameters to this function can include actual values, as in the producer of Figure 2, or any combination of actuals and formals (types), as in the consumer. The result returned by this function is a tuple that is augmented by some fields that were not present in the tuple originally written by the specifier, but that are necessary to the semantics of the operations. In formalizing Linda, the only field added by **createTuple** is a unique identifier; when formalizing LIME in the next section, **createTuple** is modified to add also fields for the current and destination location. These fields remain part of the tuple, accessible to the operations defined in our formalization, but not accessible to the specifier.

[8]This statement uses a UNITY-like three part notation of the form ⟨**op** *quantifiedVariables* : *range* :: *expression*⟩. The variables from *quantifiedVariables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for **op**, e.g., *true* when **op** is ∀.

Although we do not provide a formal definition of **createTuple**, as it is tedious and not interesting, we define here a general notation for setting and accessing these added tuple fields. The tuple identifier is associated to a *field label* ID; later on, we use the labels CUR and DEST for the current and destination location fields, respectively. These field labels are used to create template expressions. For instance, $[\text{ID: ID}] \oplus t$ sets the identifier field of tuple t to the formal ID, while $[\text{ID: } 42] \oplus t$ sets it to the actual 42. Moreover, field labels are also used to retrieve the field value. Hence, $t.\text{ID}$ returns the current location of the tuple or, in conformance with the semantics of Linda, undefined if the field is currently set to a formal.

Having augmented each tuple with an extra identifier field allows us to formalize the tuple container as a set of type TupleSpace (as opposed to a multiset), as long as the identifier value is set properly. This definition presents the specifier with constructs that allow for multiple tuples with identical data to exist in the tuple space, as in Linda, but allows us to exploit set operations in the specification of the underlying semantics.

Constructs. The final statement of the producer program demonstrates the use of the operation $\mathbf{out}(T, t)$ to insert a tuple into the tuple space. Because a process can access multiple tuple spaces, the tuple space variable appears as a parameter of the operation. The formal definition describes the tuple space change resulting from the execution of the **out** operation, namely the insertion of the tuple t:

$$\mathbf{out}(T, t) \triangleq T := T \cup \{[\text{ID: } \mathsf{newId}()] \oplus t\}$$

In addition to inserting the tuple into the tuple space, the uniqueness of the tuple is established by setting the identifier field to a system-wide unique tuple identifier, returned by the function newId(). It should also be noted that by the assignment semantics of Mobile UNITY, the value in braces is actually the value of the tuple, and thus a *copy* of the tuple is effectively made and inserted into the tuple space.

Before turning to the formal definitions of the **in** and **rd** operations for retrieving data from the tuple space, we introduce a few auxiliary definitions. First, to formally express the fact that a tuple θ matches a template p, we use the notation $\mathcal{M}(\theta, p)$. We do not provide a fully formal definition for the semantics of matching since it is not fundamental to our model. Moreover, we extend the definition of formal fields to allow the specification of subtypes. For example, $\langle \text{“foo”}, \text{Integer } i : 1 \leq i \leq 10 \rangle$ requires a matching tuple to have, in its second field, an integer value between 1 and 10. While we do not use this functionality directly in our example program, it is used in the definitions of several LIME constructs in the next subsection[9]

Next we define a predicate that identifies whether a tuple matching a pattern exists in a given tuple space, and two macros that identify a matching tuple in a

[9]Moreover, while this feature is used in the formal semantics, matching on subtypes (e.g., using the inheritance relationship) is currently not supported by the middleware.

given tuple space and either remove or copy it:

$$\begin{aligned}
\textbf{matchExists}(T,p) &\equiv \langle \exists\, \theta : \theta \in T :: \mathcal{M}(\theta, [\text{ID: ID}] \oplus p) \rangle \\
t := \mathsf{remove}(T,p) &\triangleq \langle \| \;\; \theta : \theta = \theta'.(\mathcal{M}(\theta', [\text{ID: ID}] \oplus p) \wedge \theta' \in T) \\
&\qquad :: T, t := T - \{\theta\}, \theta \rangle \\
t := \mathsf{copy}(T,p) &\triangleq \langle \| \;\; \theta : \theta = \theta'.(\mathcal{M}(\theta', [\text{ID: ID}] \oplus p) \wedge \theta' \in T) :: t := \theta \rangle
\end{aligned}$$

In both the remove and copy macros, the value of the returned tuple is bound to the variable t, leaving the pattern p with its original value. These macros model non-deterministic tuple selection by means of *non-deterministic assignment* [Back and Sere 1990]. In a non-deterministic assignment of the form $x := x'.Q$, the variable x is assigned a value x', selected non-deterministically among those satisfying condition Q. In our case, we use a similar notation to select non-deterministically a single matching tuple θ', bind it to the tuple θ, and use it to quantify the three-part notation. Because a single tuple is selected and bound, the parallel operator in the three-part notation serves the purpose of creating a quantified statement, and does not entail parallel execution of multiple statements.

With these helper functions, the definitions of the **rd** and **in** constructs follow naturally:

$$\begin{aligned}
t := \mathbf{in}(T,p) &\triangleq t := \mathsf{remove}(T,p) \quad \textbf{if matchExists}(T,p) \\
t := \mathbf{rd}(T,p) &\triangleq t := \mathsf{copy}(T,p) \quad\;\; \textbf{if matchExists}(T,p)
\end{aligned}$$

The previous definitions describe the *blocking* forms of the **in** and **rd** operations. In Linda, a process that encounters a blocking operation suspends itself until a matching tuple is found. Instead, in Mobile UNITY there is no direct notion of process blocking: statements are selected non-deterministically. Nevertheless, if a statement is selected when no matching tuple exists in the tuple space, it is equivalent to a skip. Thus, it is as if the statement to remove the tuple was *blocked* waiting for a matching tuple. In the example, when no tasks exist in the tuple space the consumer is effectively blocked waiting for the **in** operation to become enabled. With this in mind, it is possible and meaningful to put these blocking operations in parallel with other statements. For example:

$$t := \mathbf{in}(T,p) \parallel \mathit{count} := \mathit{count} + 1$$

is expected to count the number of tuples removed. However, after the macro expansion of the **in** operation, the semantics of this statement are such that each time it is selected for execution, the counter increments even if no matching tuple is taken from the tuple space. This is because only the assignment to the left of the parallel bar is inhibited until a match is found. To allow the more meaningful style of parallel assignment in which both assignments are inhibited until a match is found, we expose the **matchExists** predicate, enabling the following assignment statement with the correct counting semantics:

$$(t := \mathbf{in}(T,p) \parallel \mathit{count} := \mathit{count} + 1) \textbf{ if matchExists}(T,p)$$

It should be noted that the same parameters, namely (T,p), must be used for the **in** operation and the **matchExists** to have the desired effect.

Additional constructs. While the above constructs define the basic Linda operations, there are a number of extensions that have been shown to be useful when programming with Linda, some of which we exploit in LIME. In contrast to the blocking version of the **in** and **rd** the non-blocking operations should return ε if no matching tuple exists in the tuple space at the moment when the statement is selected. Their formal definition builds upon the blocking operations:

$$t := \mathbf{inp}(T,p) \triangleq \begin{array}{ll} t := \mathsf{remove}(T,p) & \textbf{if } \mathbf{matchExists}(T,p) \\ \| \; t := \varepsilon & \textbf{if } \neg\mathbf{matchExists}(T,p) \end{array}$$

The definition of the probing read **rdp** is identical to **inp** except the copy macro is used in place of remove, leaving the tuple space unchanged after the execution of the operation involving a successful match. The group operations that remove or copy *all* matching tuples (e.g., **ing** and **rdg**) can be formalized in a similar manner.

3.2 Formalizing LIME

In this section, we build upon the formalization presented thus far and extend it to encompass transiently shared tuple spaces and the other constructs that are peculiar to LIME. The constructs defined in the previous section, namely **out**, **in**, **rd**, **inp**, and **rdp**, are used in the formalization of the LIME constructs, but this section provides new semantics that deal with the current and destination location fields. The functions **createTuple** and **matchExists** are modified respectively to include the creation of the CUR and DEST fields and to match tuples using these additional fields. While the overall semantics remain the same, operations work with tuple spaces that are transiently, rather than permanently, shared.

Components and connectivity. In a Mobile UNITY specification, as shown in Figure 3, a program is the unit of execution and migration. In our LIME specification, a program represents the specification of the behavior of a mobile agent. In Mobile UNITY, each program component has a special location variable λ, which we use to identify the host where the mobile agent is executing. The structure of this location variable is completely application-dependent, and for our purposes it can safely be assumed to contain the IP address or the symbolic name of a host. Agents migrate by assigning a new value to this location variable, e.g., $\lambda :=$ `lime.sf.net`. We formally express connectivity among agents as a symmetric and transitive relation κ. Two agents a and b are *connected* when they are on the same host, when the hosts they reside on are directly connected, or when the hosts they reside on are transitively connected. This is expressed by the predicate $a \; \kappa \; b$. When an agent migrates, the κ relation changes to reflect the new configuration. Changes in connectivity among hosts are intentionally left outside of the specification, but nevertheless contribute to the κ relation. This is merely a consequence of choosing agents as the only active components in the system.

In applications involving physical and/or logical mobility, it is often necessary to gain information about the current connectivity of the mobile components. The natural course of action to support this need during the specification phase would be to expose the κ relation to the specifier. However, this would actually grant the specifier more power than can actually be realized in the mobile environment. For

```
System ProducerConsumer
  Program Producer(i) at λ
    declare
      jobs : TupleSpace
      jobName : {ε, A, B}
      jobInfo : String
    initially
      jobs = {} [] jobName = ε
    assign
      jobName := A  ||  jobInfo := newInfo(A)
    [] jobName := B  ||  jobInfo := newInfo(B)
    [] (out(jobs, createTuple(jobName, jobInfo))  ||  jobName = ε)
                                                  if jobName ≠ ε
    [] (out[Consumer(1)](jobs, createTuple(jobName, jobInfo))  ||  jobName = ε)
                                                  if jobName ≠ ε
    [] out(jobs, createTuple(PRIORITY, newInfo(PRIORITY)))
  end
  Program Consumer(i) at λ
    declare
      jobs : TupleSpace
      curJob, priorityTemplate : Tuple
      jobName : {A, B}
    initially
      curJob = ε [] priorityTemplate = createTuple(PRIORITY, String)
    [] isEnabled(pri)
    assign
      curJob := in(jobs, createTuple(jobName, String))     if curJob = ε
    [] curJob := ε  ||  doJob(curJob)                      if curJob ≠ ε
    [] pri :: doJob(τ) weakReaction[AgentID, AgentID](jobs,
                                                  priorityTemplate,
                                                  ONCEPERTUPLE)
  end
  Components
      Producer(0) [] Consumer(1) [] Consumer(2)
  Interactions
      ;; Transient sharing dependent on connectivity
      ⟨∀ a, b, c, T : a, b, c ∈ C ∧ T ∈ T :: a.T ≈ b.T      when a κ b
                                                  engage a.T ∪ b.T
                                                  disengage ⟨∪ c : c κ a :: c.T ↓ c⟩,
                                                            ⟨∪ c : c κ b :: c.T ↓ c⟩⟩
      ;; Migration of misplaced tuples
    [] ⟨|| θ, a, b, T : a, b ∈ C ∧ T ∈ T ∧ θ = θ'.(θ' ∈ b.T ∧ θ'.CUR = a ∧ θ'.DEST = b)
                  :: b.T := b.T − {θ} + {[CUR: b] ⊕ θ}⟩ reacts-to true
end
```

Fig. 3. A LIME producer-consumer in Mobile UNITY.

instance, consider[10] the case where an agent a queries whether agent b is connected

[10]Here and in the following we use agent names like a also to refer to the corresponding agent identifier. While this constitutes a stretch of the notation, it greatly simplifies formulas. Moreover,

to agent c by directly using the predicate $b \kappa c$. If b and c are out of range of a, there is no reasonable way for a to resolve the query about components it cannot communicate with. However, it is both useful and reasonable to allow queries about a's current connectivity state, e.g., $a \kappa b$. In our model, the latter predicate can be expressed, from the perspective of agent a, as **isConnected**(b). The use of **isConnected** effectively restricts access to the κ relation, limiting its range only to a's current connectivity context. Based on this function and the other LIME constructs described in the following, the formal semantics of the LimeSystem tuple space can be modeled straightforwardly.

Transient sharing of tuple spaces. In LIME, as opposed to Linda, identically-named tuple space variables defined in different agents must be shared transiently, *only when connectivity is available*. This is expressed in our formalization by using the sharing construct $\approx$, as in the formalization of Linda. The full form of this construct allows one to specify the condition enabling sharing (**when**), the value the shared variable assumes when this condition is established (**engage**), and the value each variable takes when the condition is falsified (**disengage**). In our case, the condition for sharing is the existence of connectivity, the engagement value is the union of the content of the tuple spaces, and the disengagement values partition the content of the shared tuple space according to tuple location and the new connectivity.

For notational convenience we define an operator, $\downarrow$, to generate a set containing all tuples from a tuple space T whose current location field refers to an agent a. Intuitively, $T \downarrow a$ selects all tuples for which a is responsible:

$$T \downarrow a \triangleq \langle \mathbf{set}\, \theta : \theta \in T \wedge \theta.\text{CUR} = a :: \theta \rangle$$

where CUR is the field label needed to access the current location field, with a notation similar to the one we used in the previous section for the tuple identifier. With this definition, transient sharing can be expressed by substituting the unconditional sharing clause in the **Interactions** section of the Linda semantics, with the following:

$$\langle \forall\, a, b, c, T : a, b, c \in \mathcal{C} \wedge T \in \mathcal{T} :: a.T \approx b.T \rangle$$

when $a \kappa b$
engage $a.T \cup b.T$
disengage $\langle \cup c : c \kappa a :: c.T \downarrow c \rangle$, $\langle \cup c : c \kappa b :: c.T \downarrow c \rangle \rangle$

where $\mathcal{C}$ and $\mathcal{T}$ have the meaning described earlier. According to this expression, the two tuple space variables named T belonging to agents a and b become shared when connectivity is established between the agents, i.e., when $a \kappa b$ is true. The shared variable T is assigned the set union of the content of the tuple spaces. Correctness of the engagement clause relies on the commutativity of the union operator to generate the same value whether $a.T \approx b.T$ or $b.T \approx a.T$, and on the fact that the tuple space is a set (rather than a multiset), thus ensuring that no tuple duplication takes place when $a.T \approx a.T$. On disengagement, i.e., when $a \kappa b$ is falsified, the content of the shared tuple space variable is partitioned such that the

the context in which the variable is used is sufficient to remove ambiguity.

contents of the tuple spaces of each agent reflect the tuples available under the new connectivity. In other words, after a disconnection the visible tuples are only those whose responsible agents are still connected to one another. In our example, the result of transient sharing is that either of the consumers can remove jobs created by the producer only when the two components are connected.

Managing misplaced tuples. In Figure 3, the **out** operation is used with a location parameter to specify that a job tuple should be migrated to the tuple space of *Consumer(1)*. While this does not guarantee that *Consumer(1)* will process the tuple, the migration makes it possible for *Consumer(1)* to process the tuple even after it is disconnected from the producer. The semantics of this extended operation can be modeled by the following:

$$\mathbf{out}[d](T, t) \triangleq \mathbf{out}(T, [\text{CUR: } a, \text{DEST: } d] \oplus t)$$

which stores the tuple t with a (the agent that executed the operation) as the current location and d as the destination. In this definition we rely on the definition of **out** given earlier for Linda, but we modify the tuple parameter by extending it with location fields. The setting of the identifier field and the change in the tuple space content is performed by the **out** described in Section 3.1, while the values of the CUR and DEST fields are set explicitly. Note that this relies on the aforementioned redefinition of the **createTuple** function to ensure the existence of these two fields. Finally, for notational convenience we allow the specifier to leave out the values for some or all locations, as shown in Table I of Section 2.2. When missing, they are assumed to be the identifier of the agent performing the **out**.

Since this statement is not guarded, it is executed independently of whether the destination d is connected to a or not. In the case where the writing and the destination agents are connected when the **out** is issued, tuple migration is immediate. In the case where connectivity does not immediately exist, the migration occurs as soon as connectivity becomes available. This opportunistic action is modeled by an additional statement that employs the reactive construct **reacts-to** provided by Mobile UNITY.

Reactive statements extend Mobile UNITY statements with the capability to specify actions that must be executed immediately after a given condition is established rather than eventually as dictated by the fair interleaving semantics. Reactive statements can appear within an individual program specification or within the **Interactions** section. The reactive statements of the entire system are logically combined to form a single reactive program that is executed to fixed point after the execution of every conventional (non-reactive) statement. It is the responsibility of the specifier to ensure fixed point of the reactive program is actually reached[11]. In our model this condition is guaranteed by construction, except for the case when a reaction generates a tuple that triggers another reaction—a condition that is nonetheless easy to analyze at the application level.

For specifying tuple migration, we define the following reactive statement in the

[11]The introduction of reactive statements is reflected in the Mobile UNITY logic, and proper inference rules are needed to prove the correctness of reactive programs. The interested reader can find details in [McCann and Roman 1998].

Interactions section:

$$\langle \| \ \theta, a, b, T : a, b \in \mathcal{C} \wedge T \in \mathcal{T} \wedge \theta = \theta'.(\theta' \in b.T \wedge \theta'.\text{CUR} = a \wedge \theta'.\text{DEST} = b)$$
$$:: b.T := b.T - \{\theta\} + \{[\text{CUR: } b] \oplus \theta\}\rangle \qquad \textbf{reacts-to } \textit{true}$$

Although connectivity is not explicitly mentioned in this formula, it is implicit that tuples can migrate from a to b only when connectivity is available. The interpretation of the above formula from the perspective of agent b shows that b selects a misplaced tuple (with current location a and destination b) from its own tuple space, $b.T$. Since $b.T$ is shared with the other tuple spaces, the presence of a tuple belonging to a inside b's tuple space guarantees that a is connected. Stated differently, a tuple cannot exist in a tuple space if the agent specified in the tuple's current location field is not present (i.e., connected). In the formula above, the misplaced tuple is then "migrated" by removing it from the tuple space, and reinserting it with the CUR field properly reset to agent b.

The semantics of reactions guarantee that tuples migrate immediately when connectivity is available. If a tuple is written while the current and destination agents are connected, the reaction fires immediately after the **out** operation. Alternately, when components holding misplaced tuples come into range, the reaction above executes in the same atomic step that engages the tuple space variables. In both cases, agents perceive an instantaneous tuple migration to the destination agent.

Formalizing location-extended query constructs. The location-extended variants of the other operations can be modeled by using the earlier defined Linda operations along with the CUR and DEST tuple fields. Here, we focus on the formalization of **in**:

$$t := \mathbf{in}[a, b](T, p) \triangleq t := \mathbf{in}(T, [\text{CUR: } a, \text{DEST: } b] \oplus p)$$

The formalizations of **inp**, **ing**, and the **rd** variants directly follow. In the above formula, the values in square brackets are the current and destination location parameters, and any of the tuple space projections defined in Section 2 can be defined easily. Agents can be named explicitly using their identifier; the formal AgentID is used to match any agent. To express queries ranging over an entire host, we use the subtype matching definition described in Section 3.1. For example, the query necessary to remove a tuple destined to agent b from the host-level tuple space named *jobs* of the co-located agent a is:

$$\mathbf{in}[\langle \text{AgentID } i : i.\lambda = a.\lambda \rangle, b](jobs, p)$$

Care must be taken to ensure that the predicate is computable given the connectivity constraints of the system. For example, it is not reasonable to write a predicate that specifies for the DEST field all agents residing on a host that is not accessible. Both predicates used here are computable, and are available in the implementation of LIME. For notational convenience, we allow the specifier to use a non-augmented version of these operations, with the default meaning to query the current federated tuple space. As discussed in Section 4.2, the CUR field of the probe and group operations should be restricted to a single host, although this is not enforced by the formalization.

Strong and weak reactions. The example in Figure 3 extends the producer and

the consumers to handle *high priority jobs*. These are generated by the producer and should be performed by any of the consumers "as soon as possible", i.e., as allowed by connectivity. If the producer and a consumer are connected when the job is generated, the consumer should immediately remove the corresponding tuple and perform the job. Otherwise, the consumer should take and perform the job as soon as connectivity is restored. From the producer's perspective, the only change is the addition of a statement (the last one) that outputs a job tuple with PRIORITY as the job name. The consumer program, on the other hand, must be able to *react* to the presence of a high priority tuple. LIME reactions provide the natural solution for implementing this behavior. As discussed in Section 2, LIME provides two kinds of reactions: strong reactions fire in the same atomic step as detecting the presence of a matching tuple, while weak reactions fire only eventually after detection. In the example, the consumer program is augmented with a statement (the last one) containing a weak reaction that is fired whenever a tuple containing high priority jobs appears in the shared tuple space, and executes the corresponding job. Using a weak reaction in the example makes it closer to the specification of a real application designed using the current LIME implementation where, as we mentioned in Section 2, performance considerations driven by the current target scenario constrain strong reactions to fire only upon detection of a local tuple. In the formalization that follows, however, we do not model explicitly the constraints the current middleware imposes on strong reactions, keeping the formal semantics of our model as free as possible from the implementation concerns specific to a given application scenario.

We model LIME reactions as:

$$\rho :: s(\tau)\ \textbf{strongReaction}[x, y](T, p, mode)$$
$$\rho :: s(\tau)\ \textbf{weakReaction}[x, y](T, p, mode)$$

where s is the statement to be performed when a tuple matching pattern p is found in the tuple space T. As in query operations, $[x, y]$ indicates a projection of the tuple space over which to evaluate the reaction. The variable τ is a free variable that can appear within the statement s and is bound to the matched tuple when the action fires. The variable *mode* can assume either of the ONCE or ONCEPERTUPLE values. The label ρ uniquely identifies the reaction and can be used as a parameter for the registering and deregistering operations, **enableReaction**(ρ) and **disableReaction**(ρ).

The formalization of reactions relies on the **reacts-to** construct provided by Mobile UNITY. Nevertheless, some additional bookkeeping is necessary to deal with explicit registration and deregistration of reactions (which is not provided in Mobile UNITY) and to ensure the proper execution pattern with respect to the reaction mode. The details of the formalization are discussed below for strong

reactions:

$$\rho :: s(\tau)\ \textbf{strongReaction}[x, y](T, p, mode) \triangleq$$
$$\langle \| \ \theta : \theta = \theta'.(\mathcal{M}(\theta', [\text{ID: ID}, \text{CUR: } x, \text{DEST: } y] \oplus p) \wedge \ \theta' \in T$$
$$\wedge \ (mode = \text{ONCE} \Rightarrow reactedSet_\rho = \{\})$$
$$\wedge \ (mode = \text{ONCEPERTUPLE} \Rightarrow \ \theta'.\text{ID} \notin reactedSet_\rho))$$
$$:: s\{\tau/\theta\} \ \| \ reactedSet_\rho := reactedSet_\rho \cup \{\theta.\text{ID}\}\rangle$$

reacts-to

$$\textbf{matchExists}(T, [\text{ID: ID}, \text{CUR: } x, \text{DEST: } y] \oplus p) \wedge enabled_\rho$$

To keep track of the tuples that have been reacted to and of whether the user has enabled or disabled the reactions, two auxiliary variables, namely $reactedSet_\rho$ and $enabled_\rho$, are introduced. By subscripting these variables with the unique reaction identifier ρ, we are guaranteed that reactions do not interfere with one another.

The selection of a matching tuple uses the same non-deterministic selection notation as described in the Linda formalization for the copy function. The necessity to bind the matched tuple to the free variable τ prohibits the use of the copy function here, but the semantics are the same: a single matching tuple is selected from the shared tuple space. The condition for actually firing the reaction (i.e., executing the user action s) depends on the contents of $reactedSet_\rho$. This variable tracks the identity of the tuples that have already been reacted to. A ONCE reaction will only fire if no tuples have been reacted to, i.e., the set is empty. A ONCEPERTUPLE reaction will only fire if the tuple chosen by the non-deterministic selection has not been recorded in $reactedSet_\rho$. The user action $s(\tau)$ can be any non-reactive Mobile UNITY statement[12].

To guarantee that at least one tuple can be selected from the tuple space, we include the **matchExists** function as a condition to the reaction. Without this, the θ' variable could not be bound and the formalization would be incorrect. The second condition for the reaction, $enabled_\rho$, tracks whether the user has explicitly enabled or disabled the reaction with the following macros:

$$\textbf{enableReaction}(\rho) \ \triangleq \ (enabled_\rho := true \ \| \ reactedSet_\rho := \{\}) \quad \textbf{if } \neg enabled_\rho$$
$$\textbf{disableReaction}(\rho) \ \triangleq \ enabled_\rho := false$$

These functions allow the specifier to control when reactions are able to fire during the lifetime of the system. By clearing $reactedSet_\rho$ when a reaction is re-enabled, it is treated as a new reaction that has not fired on any tuples. This is true for both ONCE and ONCEPERTUPLE; neither type of reaction retains any *memory* of previous periods when it was enabled and they may react again to the same tuples, if re-enabled. It is also important to remember that the reactive program is executed after every regular statement, including a statement that enables a reaction. This means that a reaction may fire in the same atomic step as the statement that enables it, assuming a matching tuple exists in the tuple space. As mentioned in Section 2, this is a significant departure from more conventional event-based systems.

[12]More precisely, transactions cannot appear in user actions either. Since transactions are not used in our formalization of LIME, we redirect the reader interested in the semantic details of Mobile UNITY to [McCann and Roman 1998].

For proper initialization, each reaction must be either enabled with an empty $reactedSet_\rho$, or disabled. This is indicated in the **initially** section of a program for reaction ρ by using either of the following predicates:

$$\textbf{isEnabled}(\rho) \triangleq enabled_\rho = true \wedge reactedSet_\rho = \{\}$$
$$\textbf{isDisabled}(\rho) \triangleq enabled_\rho = false$$

As mentioned earlier, the formalization just introduced does not limit the scope of strong reactions to the local host, as is done by our current implementation. This constraint is straightforward to capture, but we prefer to define here the formal semantics of the LIME model in the most general way.

Moreover, our model provides, through weak reactions, an alternative reaction semantics that imposes fewer implementation requirements. Weak reactions remove the atomicity constraint between the detection of the matching tuple and the firing of the reaction, allowing other agent operations to occur in between the two. Weak reactions therefore enjoy a scope that can be as large as the whole federated tuple space without excessive constraints being placed on the run-time system. We take advantage of the similarities between weak and strong reactions and actually rely on the latter when specifying the former. Indeed, a strong reaction is exploited to detect the appearance of a matching tuple. The user action is instead an asynchronous statement, breaking the atomicity and guaranteeing only eventual completion of the weak reaction:

$$\begin{aligned}\rho :: s(\tau)\ \textbf{weakReaction}[x,y](T,p,mode) \triangleq \\ \rho :: events_\rho := events_\rho \cup \{\tau\}\ \textbf{strongReaction}[x,y](T,p,mode) \\ [\!] \ \langle \| \ \theta : \theta = \theta'.(\theta' \in events_\rho) :: events_\rho := events_\rho - \{\theta\} \ \| \ s\{\tau/\theta\}\rangle\end{aligned}$$

To model the transfer of a matching tuple from a given agent to the subscriber, we rely on the set $events_\rho$. This variable is a temporary holding place for all tuples that should be reacted to, but for whom the user reaction has not yet fired. In some sense, even though $events_\rho$ is local to the agent that registered the reaction, it is analogous to a communication buffer between the agents. This set is populated by the strong reaction in the first statement. The second statement, separated by the $[\!]$ operator, executes asynchronously with respect to the reaction by removing an element from $events_\rho$ and firing the corresponding user action using the matched tuple. As with strong reactions, only tuples in the currently shared tuple space will be reacted to, meaning that any tuples written while hosts are disconnected cannot be reacted to, even in the weak model. Additionally, we must redefine the initialization macro **isEnabled**, used only in the **initially** section, to clear the set $events_\rho$:

$$\textbf{isEnabled}(\rho) \triangleq enabled_\rho = true \wedge reactedSet_\rho = \{\} \wedge events_\rho = \{\}$$

One point to notice about the definition of **weakReaction** is the use of non-deterministic selection to remove an element from $events_\rho$. Similar to the definition of remove in Section 3.1, exactly one tuple is selected, ensuring that only one user action s fires at a time. Also, this non-deterministic selection does not guarantee an order in the selection of elements from $events_\rho$, nor does it guarantee that an element will *ever* be selected. This is the weakest constraint we can set in the

```
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();
}
```

Fig. 4. The class `LimeTupleSpace`, representing a transiently shared tuple space.

formalization, and leaves room for stronger guarantees to be enforced by specific implementations. For instance, our current implementation maintains ordering and guarantees selection.

4. THE LIME MIDDLEWARE

After having presented the LIME model and its formal semantics, we now switch our focus to describe how it is embodied in the companion middleware. In Section 4.1, we illustrate how the concepts of our model are made available to the programmer through the middleware application programming interface (API). Then, in Section 4.2 we discuss the most important choices underlying the design of the LIME middleware, and illustrate its overall architecture. Finally, Section 5 describes some examples of application development, where we illustrate how the API is used in practice by the programmer, and how the mechanisms included in the middleware architecture come into play in mobile scenarios. The reader interested in additional information can find extensive documentation and programming examples on the LIME Web site at `http://lime.sourceforge.net`.

4.1 Application Programming Interface

Fundamental to LIME is the notion of transiently shared tuple space. This concept is embodied in the class `LimeTupleSpace`, whose public interface is shown[13] in Figure 4. 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. The name of the tuple space is specified as a parameter of the constructor.

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

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 4). 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 configuration properties, and so on.

`LimeTupleSpace` contains the Linda operations needed to access the tuple space, as well as their 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 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 and therefore including 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 (`inp` and `rdp`) and bulk operations (`rdg` and `ing`) are always restricted to a local subset of the federated tuple space, as defined by the location parameters. An unconstrained definition, such as the one provided for `in` and `rd`, would involve a distributed transaction across the federated tuple space to preserve the semantics of the probe.

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

```
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);
}
```

Fig. 5. The classes `Reaction`, `RegisteredReaction`, `ReactionEvent`, and the interface `ReactionListener`, required for the definition of reactions on the tuple space.

generate a run-time exception when invoked on a private tuple space.

The remainder of the interface of `LimeTupleSpace` is devoted to managing reactions; relevant classes for this task are shown in Figure 5. 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 instance of `RegisteredReaction` that can be used to deregister a reaction with the method `removeReaction`. The decoupling between the reaction used for the registration

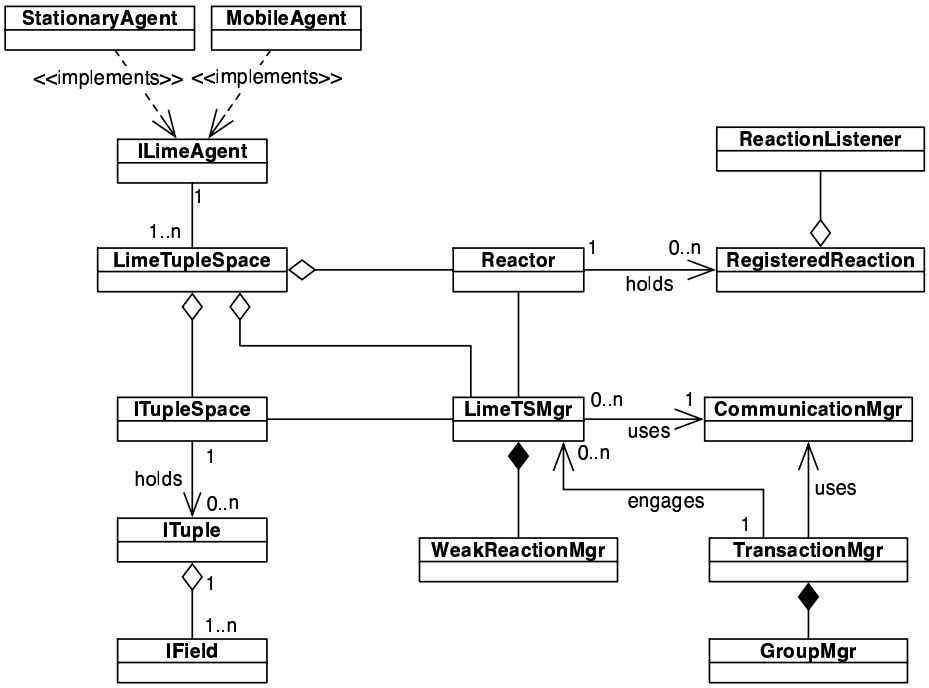

Fig. 6. The main components of the LIME middleware architecture.

and the returned `RegisteredReaction` object 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.

Finally, in addition to the `LimeTupleSpace` class described thus far, the LIME API includes a `LimeSystemTupleSpace` class that provides access to the LimeSystem tuple space. Since tuples in the LimeSystem are directly managed by the run-time support, `LimeSystemTupleSpace` does not provide any of the variants of the **out** or **in** operations. The interface of this class is otherwise identical to `LimeTupleSpace`, enabling queries and reactions over the system configuration information.

4.2 Design and Implementation

In this section we look behind the scenes of the LIME API, and describe the main design choices of our middleware. Figure 6 shows a class diagram with the most relevant classes and their relationships. Details are provided throughout this section.

Tuple Space. One of the early decisions in the design of LIME addressed the implementation of the underlying tuple space. Analysis of available systems revealed that they provide a rich set of features with large variations in terms of expressiveness, performance, and often semantics. The need for a simple, lightweight implementation, combined with the desire to provide support and interoperability

with industrial-strength products, led us to the development of an adaptation layer that hides from the rest of the LIME implementation the nature of the underlying tuple space engine. This layer is provided by a separate package called LIGHTS that also includes a lightweight tuple space implementation[14]. In this package, the interface `ITupleSpace` provides access to the core tuple space functionality. Adapter classes implementing this and other interfaces are loaded at start-up to translate operations into those of the supported tuple space engines. Currently, adapters exist for LIGHTS's built-in lightweight tuple space and for IBM's TSpaces [Lehman et al. 2001].

In LIME, every `LimeTupleSpace` object contains an instance of the LIGHTS tuple space, accessible through the `ITupleSpace` interface. Instances of `LimeTupleSpace` are created as private, so that only the agent creating the tuple space has access to the contents of the tuple space. Operations on the `LimeTupleSpace` are delegated to the underlying `ITupleSpace` object if the tuple space is currently private. Otherwise, they are delegated to a component of the run-time that enforces the semantics of transient sharing, as described in the following.

Location Parameters. Although locations are not immediately useful in a private tuple space, `LimeTupleSpace` is equipped with the mechanisms needed to deal with location attributes since they become relevant when the tuple space is toggled from private to shared. These mechanisms simply entail the management of two fields that are dynamically added and removed from tuples, and that correspond to the current and destination locations. Upon insertion in the tuple space, the original tuple provided by the programmer is always augmented with these two fields, which are set to the proper value—those specified by the programmer if the operation contains location parameters, conventional default values otherwise. These fields are then stripped off when the tuple is returned as the result of an operation accessing the tuple space, such as **rd**. Although these fields are exploited by the run-time support to deal with locations, they are hidden from the programmer, thus preventing the introduction of inconsistencies by directly altering their values.

These location parameters are simply agent identifiers. It is worth noting that agent identifiers must be unique throughout the system. For this reason, every agent is assigned a unique identifier indirectly accessible through the `ILimeAgent` interface. The agent identifier itself contains the identifier of the server where it was created, and a unique `long` value, incremented each time a new agent is created on the server.

Host-Level Sharing. When a private `LimeTupleSpace` is set to shared, a host-level tuple space is created through transient sharing, according to the semantics we presented earlier. The enforcement of these semantics, and in particular of the engagement and disengagement of local tuple spaces, cannot be managed by a single `LimeTupleSpace`, as it requires host-wide management of tuple space access. This management is provided by instances of the class `LimeTSMgr`. On each host, one `LimeTSMgr` object exists per tuple space name[15]: each of these objects is created

[14] LIGHTS source code and documentation are available at `lights.sourceforge.net`

[15] We assume applications use some convention for tuple space naming. Alternately, the `LimeSystemTupleSpace` can be used to identify the names of the tuple spaces in the system.

when a local `LimeTupleSpace` with the corresponding name becomes shared for the first time. Subsequent engagements of local tuple spaces with the same name exploit the same `LimeTSMgr`.

When the `LimeTupleSpace` becomes shared, it surrenders the control of its tuple space to the appropriate `LimeTSMgr`. In practice, the methods of the `LimeTupleSpace` no longer operate directly on the `ITupleSpace`, rather they delegate tuple space operations to the `LimeTSMgr`, which performs the appropriate actions according to the semantics of transient sharing before returning control to the method body and hence to the calling agent. Operation requests are queued and serially executed at the `LimeTSMgr`, which runs in a separate thread of control. This way, synchronization among concurrent accesses performed through different `LimeTupleSpace` instances is obtained structurally, by confining all tuple space accesses to a synchronized queue.

To perform its actions, the `LimeTSMgr` must obtain access to the content of the tuple space being shared. This access can be granted in at least two ways. The first one consists of providing the `LimeTSMgr` with a direct reference to the `ITupleSpace` object holding such content. The other solution consists of moving the data tuples into a single `ITupleSpace` that contains all the shared tuples and is associated with the `LimeTSMgr`, and moving them back to the original `ITupleSpace` when it is unshared[16]. The first approach is more convenient in very dynamic scenarios, where the overhead of moving the tuples back and forth as a consequence of a reconfiguration becomes significant. Instead, the second approach, which is chosen by our current implementation, opens up opportunities for optimizing query execution since, rather than searching several tuple spaces, a single one can be searched. It should be noted that in both cases, the tuples are *moved*, not copied.

Throughout this section, we use the term *host* to refer to the location of a group of agents. In LIME, however, a single computer can host multiple groups, distinguished from one another by port number. Technically, therefore, when we use the term host we are referring to a `LimeServer`, identified through a `host:port` pair.

Reactions. In addition to the `ITupleSpace` object, each `LimeTupleSpace` and `LimeTSMgr` contains also a `Reactor` object that is responsible for managing the (de)registration of reactions and the execution of the reactive program. `Reactor`, `LimeTupleSpace`, and `LimeTSMgr` are properly synchronized so that only tuple space operations issued from within the statements of the reactive program are allowed to execute while all others are blocked during the execution of a reaction.

When a strong reaction is registered by the user, a `RegisteredReaction` object is generated and kept in a list held by the local `Reactor`. This list effectively contains the reactive program, which must execute after each tuple space operation that adds tuples to the tuple space. Execution of the reactive program proceeds by iterating through the list of registered reactions. If a tuple matching a reaction is found, the reaction is fired by executing the corresponding `ReactionListener`. Iteration continues until one pass completes with no reactions firing.

Weak reactions are built on top of strong reactions, and may span multiple hosts.

[16]In this case, the tuple space associated with the `LimeTSMgr` can be regarded as a concrete representation of the host-level tuple space.

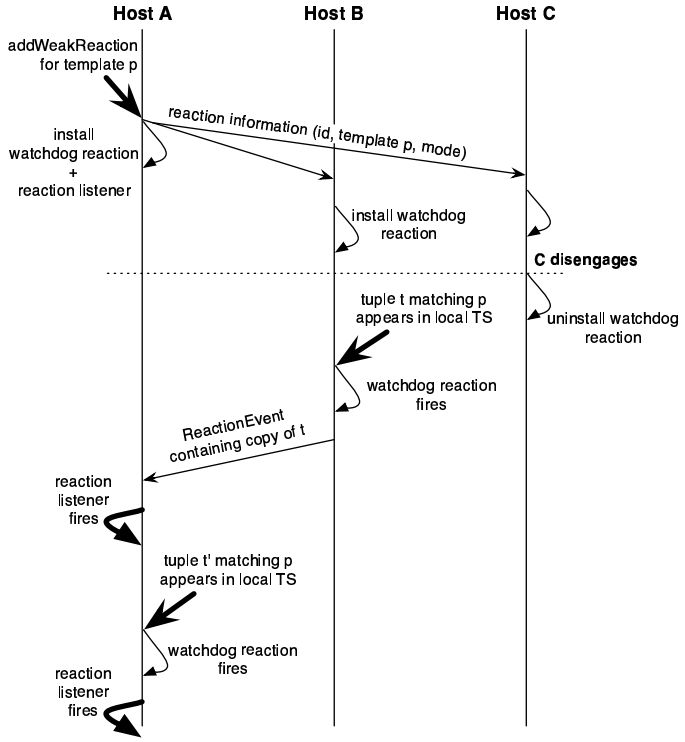

Fig. 7. Distributed processing of a weak ONCEPERTUPLE reaction.

Figure 7 shows an example of the distributed processing involved. When a weak reaction is registered (e.g., in the figure, by an agent in host A), a message containing the reaction template and a reaction identifier is sent to all the hosts involved, based on the location parameters associated with the reaction. On each host, the arrival of this message causes the registration of a system-defined strong reaction, effectively watching for tuples matching the specified template on behalf of the registering agent. A similar "watchdog" reaction is registered also at the host of the agent that installed the weak reaction. Moreover, on the registering host, the `ReactionListener` provided by the programmer is stored along with the reaction identifier within a table managed by the `WeakReactionMgr` object associated to the local `LimeTSMgr`. When one of the watchdog strong reactions fires on one of the hosts (e.g., because of the insertion of a matching tuple in the local tuple space, as in Figure 9), its listener takes care of sending back to the registering host the corresponding `ReactionEvent`, which contains a copy of the matching tuple and the reaction identifier. The latter is used to retrieve the correct listener from the table in `WeakReactionMgr` and to execute it by passing the `ReactionEvent`.

Upon disengagement, the watchdog system reactions are automatically deregistered, as shown in Figure 9 with the disengagement of host C. Otherwise, these reactions may or may not remain installed after the execution of the listener on

the registering agent, depending on the mode of the application reaction installed. If the mode is ONCE, the reaction is deregistered (i.e., removed from the reactive program, and its listener removed from `WeakReactionMgr`) right after its execution, and so are all the system reactions. We guarantee that the `ReactionListener` is executed only once, even when multiple matching tuples are returned from different hosts in the system.

Instead, if the mode is ONCEPERTUPLE the reaction (and the remote watchdog reactions necessary for monitoring) remains registered, but it must be prevented from firing multiple times for the same tuple. This requires a mechanism to distinguish one tuple from another. Because the tuple space itself is a multiset, multiple tuples may contain the same data. Therefore, we guarantee that all tuples are unique by adding a tuple identifier field to each tuple before it is inserted into the tuple space. To guarantee uniqueness, this tuple identifier contains the identifier of the agent that produced the tuple and a `long` value that is incremented each time a tuple is inserted by the agent. The identifier is hidden from the LIME programmer in the same manner as the location parameters, meaning it is stripped from the tuple before the data portion of the tuple is returned to the user.

In the current implementation, the ONCEPERTUPLE mode is implemented by keeping a list of the tuple identifiers that have already been reacted to within the `RegisteredReaction` object itself. Each time a matching tuple is found, this data structure is queried and updated to determine if a listener should be executed. Our optimized implementation of `Reactor` separates newly written tuples from those that were in the tuple space prior to the execution of a given reaction, and greatly improves the performance of ONCEPERTUPLE by not selecting a tuple more than once from a single local tuple space. However, because tuples can migrate and weak reactions can be uninstalled and reinstalled as connectivity changes, it is possible for a tuple to be selected more than once for a reaction, making the list of tuple identifiers necessary.

Reactions as a Building Block of Transiently Shared Tuple Spaces. One interesting aspect of our design is that the management of blocking operations over a transiently shared space, including a federated one, is performed by relying on reactions, which are exploited not only by the programmer through the API, but also internally by the run-time support.

When a blocking query operation is issued on a federated tuple space, one of two things may happen. In the case where a matching tuple is found immediately in the host-level tuple space, the processing is equivalent to the non-blocking operation: the `LimeTSMgr` simply releases the calling agent and returns the matched tuple to it. Instead, if no matching tuple is found, further processing must be done to detect when a matching tuple appears in the tuple space, and therefore the agent can be released. Phrased another way, the run-time support needs to *react* to the presence of a matching tuple, an operation that is achieved precisely by using LIME reactions.

Therefore, when the immediate probe does not find a matching tuple, the run-time creates and registers a ONCE reaction with the same template. When a matching tuple is eventually inserted in the tuple space, the reaction fires and the tuple is passed to the listener through the `ReactionEvent` input parameter—as in all

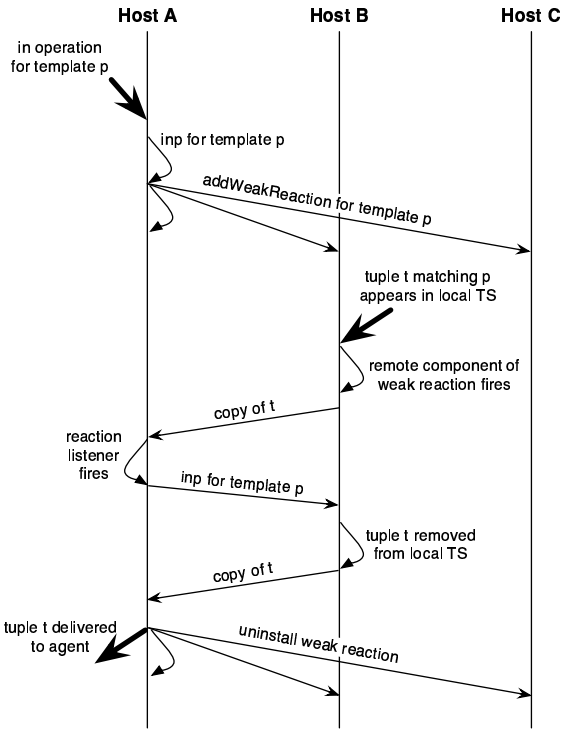

Fig. 8. Distributed processing of an **in** operation. The example assumes no matching tuple initially exists at Host *A*.

reactions. The listener is system-defined, and its body performs different actions depending on whether the operation originally requested was a **rd** or **in**. For a **rd**, the matching tuple is passed immediately to the suspended agent. For an **in**, the tuple is first removed, then passed to the agent. This processing is shown in Figure 8.

These operations take place both in the case of a host-level and a federated tuple space, with the only difference being that in the first case a strong reaction is registered, while in the second case a weak reaction is used. This complicates things when an **in** must be processed. In fact, in the first case the use of a strong reaction guarantees that the tuple cannot be withdrawn in between the detection of its presence and the execution of the listener, since these two steps are executed atomically. This guarantee does not hold when a weak reaction is used, and hence a subsequent **inp** must be used to withdraw the tuple. Probes are implemented by sending the **inp** operation request to the appropriate host based on the location parameters. There, it is served by the appropriate `LimeTSMgr` and the result is returned. In our case, if the probe returns a tuple the agent is released. However, the tuple might have been withdrawn by another agent, in which case the **inp** returns `null`, the reaction is re-registered, and the agent continues to wait.

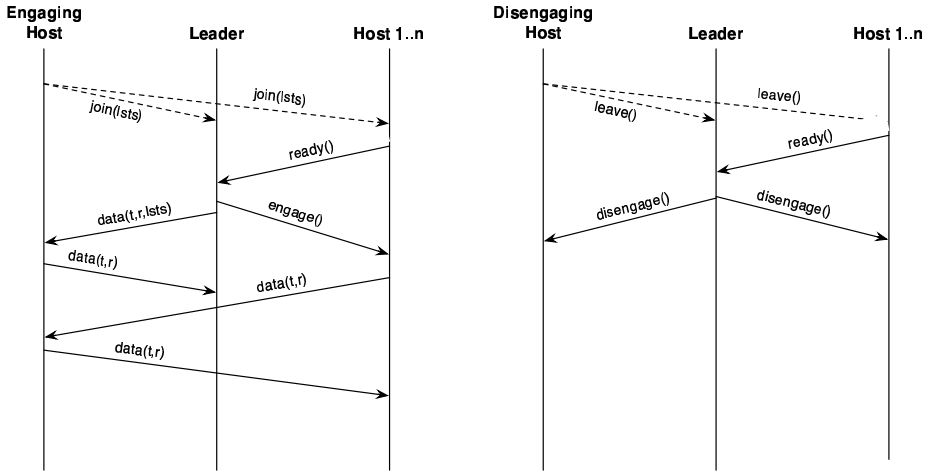

Fig. 9. A simple leader-based protocol for engagement and disengagement. The leader serves to ensure atomicity of the operation.

Another case to consider is the possibility that the reaction fires at the same time on multiple hosts, with the result that the host of the agent that issued the operation receives multiple matching tuples. If the operation is a **rd**, the agent's host simply returns to it the first tuple, dropping any others that arrive. If the operation is an **in**, the replies are buffered and if the **inp** for the first tuple returns `null`, the subsequent replies are used to issue new **inp** requests to find a matching tuple.

Connectivity, Engagement, and Disengagement. Transient sharing of host-level tuple spaces into a federated one is dependent on the connectivity among the hosts in the system. A set of connected hosts forms a *group*. Independent, disjoint groups can co-exist, as long as they are not within communication range. When two or more groups move within range, they are merged into a single group containing all members. To provide this functionality, LIME relies on the group management protocol described in [Roman et al. 2001]. This protocol guarantees that, in response to a change in group membership detected at the network level, all members of the group are notified atomically. In principle, any group formation policy can be exploited, but the current implementation matches the above description and relies on GPS information, working best in outdoor scenarios.

To give the reader a basic understanding of how the engagement protocol works, we describe here a simpler engagement protocol based on multicast instead of GPS. This protocol actually served as our first experimental engagement protocol and is still available in the current implementation to support indoor scenarios where GPS is not available and nodes announce explicitly connection and disconnection. It assumes that hosts are added to and removed from groups one at a time and does not support group-to-group engagement or disengagement. The operation of this engagement and disengagement protocol is shown in Figure 9.

For each LIME group, a single node acts as the leader, ensuring the atomicity of engagement. Before a node can join, operations on all involved local tuple spaces must be halted. This is accomplished by the new host multicasting a join request. Upon receipt of this request, hosts stop local processing and inform the leader that they are now ready for engagement. When all nodes are ready, the leader informs the new host of the identity of the group members by sending a copy of its LimeSystem tuple space and tells all group members to begin the engagement. All group members, including the leader, then exchange misplaced tuples and remote reactions with the new host. When all pairwise data exchanges have completed, regular operation continues at the hosts. The new host identifies completion when it has received data from all hosts identified by the leader's LimeSystem tuple space.

Disengagement is similar, with the departing node notifying the leader of its intent to depart. As before, all nodes receive this message, complete any outstanding communication with the departing host and notify the leader they are ready for disengagement. When all hosts are ready, the leader announces disengagement and the host has disengaged.

In both versions of the engagement and disengagement protocols, the LimeSystem is updated to reflect the new configuration of hosts, agents, and tuple spaces. In Figure 6 this is explicitly shown with the connection between `GroupMgr` and the `TransactionMgr`. The `TransactionMgr` ensures the atomicity of the engagement and disengagement.

It should be noted that in the implementation, the LimeSystem is not actually implemented as a federated tuple space. Instead it is independently maintained at each host, and is kept consistent though updates during engagement and disengagement. Besides updating the information inside the LimeSystem tuple space, engagement effects the migration of all the misplaced tuples whose destination became available during the group change. Moreover, for all the weak reactions involving newly connected hosts, engagement also triggers the registration of these reactions on the appropriate hosts. On the other hand, disengagement does not involve any transfer of information. When hosts are disconnected as a consequence of a group change, the weak reactions involving them are simply deregistered and the LimeSystem updated. No movement of tuples is required although, to preserve atomicity, the protocol we employ performs a distributed transaction to ensure that all hosts complete the disengagement process before regular operations resume.

Mobile Agents. LIME is a coordination framework that deals with mobility in a way that is independent from the nature of migration. As this is not one of its goals, it does not support directly agent mobility. 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 adapter built for the μCODE mobile code toolkit [Picco 1998], available as open source at `mucode.sourceforge.net`.

This adaptation layer allows a mobile agent to carry along one or more LIME tuple spaces, and automatically deals with their (dis)engagement. Upon 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 they are deserialized and shared again before the agent code

begins to execute. More details about the adaptation layer and how to integrate a different mobile agent system are available in the LIME documentation, and on the LIME Web site.

Implementation Details. The `lime` package is roughly 5,000 non-commented source statements, resulting in approximately 100 Kbyte of `jar` file. The LIGHTS lightweight tuple space implementation and the adapter for integrating multiple tuple space engines add an additional 20 Kbyte of `jar` file. When using mobile agents, the μCODE toolkit adds approximately 30 Kbyte of `jar` file. Communication is completely handled at the socket level, requiring no support for RMI or other communication mechanisms. LIME has been tested successfully on various versions of Windows, Linux, and MacOSX, using wired Ethernet as well as IEEE 802.11 wireless technology. Moreover, LIME was tested successfully also on PDAs equipped with PersonalJava and other Java environments for small devices.

5. DEVELOPING MOBILE APPLICATIONS WITH LIME

Application development is the last phase of our research strategy, and the one where the abstractions inspired by formal modeling and embodied in the middleware are evaluated against the needs of practitioners. In this section we present two applications that exploit the current implementation of LIME in a setting with physical mobility of hosts. The first involves the ability to perform collaborative tasks in the presence of disconnection, while the second revolves around the ability to detect changes in the system configuration. In each case, we present the corresponding application scenarios and report how LIME has been exploited during development. The lessons learned from these experiences and the results of our empirical evaluation of LIME are presented in the next section.

5.1 ROAMINGJIGSAW: Accessing Shared Data

Scenario. Our first application, ROAMINGJIGSAW, is a multi-player game in which a group of players, each carrying a PDA, cooperate to assemble a jigsaw puzzle. They can construct assemblies of two or more pieces independently (e.g., while disconnected), acquire piece descriptions from one another when connected, and share intermediate results (e.g., parts of the puzzle already assembled). Play begins with one player loading the puzzle pieces into a shared workspace that is visualized by the user as a *puzzle tray*. The workspace is shared among all connected users, therefore the puzzle trays of all connected users show the same set of puzzle pieces at the beginning.

Each player is assigned a unique color, and players are restricted to working only with pieces outlined with their color. Pieces can be selected by clicking on them, resulting in a change of the outline color, visible on the displays of all users. A player can select pieces or assemblies that are currently selected by another player, provided that the target player is connected.

Disconnection of a player does not have an immediate effect on the view of the puzzle tray of the others. Nevertheless, pieces that have been selected by the departing player can no longer be selected by the others—and vice versa. Hence, the disconnected player can now construct assemblies by using only the pieces out-

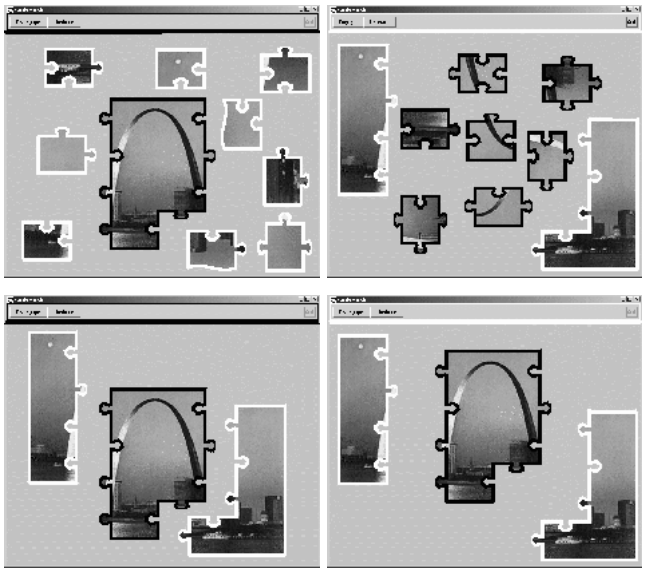

Fig. 10. RoamingJigsaw. The top two images show the puzzle trays of the black and white players while they are disconnected and able to assemble only their selected pieces. The bottom two images show the black and white puzzle trays after the players re-engage and see the assemblies that were made during disconnection.

lined with her color. The pieces of all players remain visible, but if any of the disconnected players assembles pieces, these will not be visible due to the disconnection. These assemblies become visible only when connectivity is restored. In fact, upon reconnection, the puzzle trays of all users are reconciled with all changes made during disconnection and the selection of a piece belonging to a connected player is again possible. Figure 10 shows the appearance of the puzzle tray during disconnection and after reconnection.

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, even this weak model allows players to work towards the global goal, i.e., the solution of the puzzle, through incremental updates of their local state.

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 is 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.

```
public class LimePuzzleAgent extends StationaryAgent implements ReactionListener{
  public LimePuzzleAgent() throws LimeException { super(); }
  public void run() {
    opQueue = new Queue(); // initialize Lime operation queue
    PuzzleTray puzzleTray = new PuzzleTray(opQueue); // create GUI
    LimeTupleSpace lts = new LimeTupleSpace(''puzzleName'');
    lts.setShared(true);                                                          (1)
    AgentLocation thisAgent = new AgentLocation(this.getMgr().getID());
    ITuple pieceTemplate = new Tuple().addFormal(Integer).addFormal(BitMap);
    UbiquitousReaction ur =
      new UbiquitousReaction(pieceTemplate, this, Reaction.ONCEPERTUPLE);         (2)
    lts.addWeakReaction(new Reaction[]{ ur });
    while (true) {
      Queue op = opQueue.remove(); // block on the next operation
      switch(op.type) {
      case NEW_PUZZLE:
        for(Enumeration e = op.getPieces().elements(); e.hasMoreElements(); ) {
          PuzzlePiece p = (PuzzlePiece) e.next();                                 (3)
          lts.out(new Tuple().addActual(p.getPieceID()).addActual(p.getPixels()));
        }
        break;
      case SELECT_PIECE:
        ITuple t = lts.inp(op.getPieceOwner(), AgentLocation.UNSPECIFIED,
                           new Tuple().addActual(op.getPieceID()).addFormal(Bitmap));
        if (t==null)
          /* beep to notify that select failed */                                 (4)
        else {
          lts.out(t);
          puzzleTray.update(t, thisAgent); // change the outline color of the piece
        }
        break;
      case ASSEMBLE_PIECES:
        ITuple p1 = lts.inp(thisAgent, AgentLocation.UNSPECIFIED,
                            new Tuple().addActual(op.getPieceOne().getPieceID())
                                       .addFormal(BitMap));
        if (p1==null)
          /* beep to notify that assembly failed */
        else {
          ITuple p2 = lts.inp(thisAgent, AgentLocation.UNSPECIFIED,
                              new Tuple().addActual(getPieceTwo().getPieceID())
                                         .addFormal(BitMap));
          if (p2==null)                                                           (5)
            /* beep to notify that assembly failed */
            lts.out(p1);  // put back the first puzzle piece
          } else {
            ITuple t = new Tuple().addActual(op.getPieceOne().getPieceID())
                                  .addActual(mergeBits(p1,p2));
            lts.out(t);
            puzzleTray.update(t, thisAgent); // display merged piece
          }
        }
        break;
      }
    }
  }
  void reactsTo(ReactionEvent re) {
    puzzleTray.update(re.getEventTuple(), re.getSourceAgent());                   (6)
  }
}
```

(1) create Lime tuple space, share it, create location object for this agent

(2) create template <id, image of connected pieces>,
create ubiquitous reaction with it, register as a weak reaction

(3) create new puzzle: output a tuple <id, bitmap> for each puzzle piece

(4) select piece: remove piece tuple from previous owner,
insert it in the local tuple space

(5) piece assembly: attempt removal of the two pieces;
if successful, output tuple containing the assembly

(6) reaction listener: reacts to the presence of a tuple by displaying the
piece image (in the tuple) outlined by the user color (based on the source agent)

Fig. 11. Code for the ROAMINGJIGSAW agent. Exceptions are not shown for readability.

Design and Implementation. Our design of RoamingJigsaw represents pieces and assemblies as tuples and the shared workspace as a tuple space. When a player creates a new puzzle, a tuple is created for each piece. When a player selects a piece, the corresponding tuple is withdrawn and subsequently reinserted in the tuple space, making itself the current owner of the tuple. Similarly, when a player builds an assembly out of several pieces, the tuples representing the previously individual pieces are removed and a new tuple is written containing information about the assembled pieces. Figure 11 shows the code for a RoamingJigsaw agent. The code shown is *real*, minus exceptions and user interface code. As the reader can appreciate, despite the non-trivial nature of the puzzle agent the code itself is reasonably clean and concise.

The critical issues in the design of RoamingJigsaw are the detection of piece selection and assembly, the reconciliation of the puzzle tray taking place on reconnection, and the joining of a new player. Interestingly, all of these rely upon a *single* weak, ubiquitous, oncepertuple reaction. The reaction template specifies any new tuple corresponding to a puzzle piece, while the reaction listener takes care of updating the puzzle tray using information found in the tuple, thus correctly maintaining the weakly consistent view of the workspace. The body of the reaction is shown in step 6 at the bottom of Figure 11 while the installation is in step 2. Since the reaction is scoped over the whole federated tuple space, the RoamingJigsaw agent receives updates about new pieces regardless of where and why they have been inserted. For example, a tuple is immediately reacted to upon insertion if it represents an assembly made by a currently connected player. If instead it is an assembly inserted by a player previously disconnected, it is immediately reacted to as soon as connectivity is re-established. It is important to notice that the Lime puzzle agents are not explicitly aware of the arrival and departure of players, thus the programming effort is rightfully spent on handling data changes, rather than monitoring changes in the system configuration.

Although the processing described thus far operates on the federated tuple space, fine-grained control over the location of tuples is critical in dealing with disconnections. To ensure that a player can access her selected pieces during a period of disconnection, piece selection must actually transfer the corresponding tuple into the local tuple space of the player's application. Moreover, according to the earlier discussion, a player must be prevented from selecting a piece that is currently not present in the federated tuple space. For this reason, as shown in step 4 of Figure 11, selection is performed by issuing an **inp** operation on the tuple space of the player last known to have the piece. If the piece is returned, it is reinserted in the local tuple space of the new owner, thus leading to a successful selection. Instead, if no tuple is returned, this means that the piece is unavailable for selection, and the game beeps in warning.

Given this ability for a player to remove another player's piece at any time, it could happen that a player tries to assemble two of its own pieces but, in the mean time, one of these pieces is removed by another player. This is handled in step 5 of Figure 11. If either of the two pieces involved in the assembly cannot be found, then the system beeps and restores the tuple space. This kind of high-level transaction is easily handled by the application.

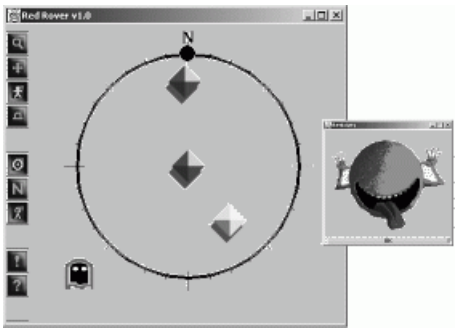

Fig. 12. REDROVER. The main console of REDROVER, the identifying icon of one of the team members, and one player (a ghost) out of range.

5.2 REDROVER: Exploiting Context-Awareness

Scenario. Our second target application is a spatial game we refer to as REDROVER, in which individuals equipped with small mobile devices form teams and interact in a physical environment augmented with virtual elements. This forces the participants to rely to a great extent on information provided by the mobile units and not solely on what is visible to the naked eye. Our final aim is to empower each player with global positioning system (GPS) access, audio and video communication, range finding capabilities, and much more. Currently, the game is limited to seeking the physical flag of another team and gathering around the player who finds the flag. A snapshot of the graphical user interface is shown in Figure 12. The most dominant display element is a view of the playing field, indicating the current position of all players within range. In ROAMINGJIGSAW, disconnection is masked until a user tries to access a piece from a disconnected user. In REDROVER disconnection is instead made explicit, by displaying a "ghost" icon for a disconnected user, indicating her temporary unavailability. Each player has an icon of herself that is available to her teammates, as shown in Figure 12. While this is currently a static image, the functionality can easily be extended to share recent images taken from a digital camera in order to share environmental information among teammates. It is important to note that while location information should be disseminated to all players, some data should be restricted only to team member access. For example, it is desirable to inform only one's own team members regarding the flag location, and not the opponents' team.

As with ROAMINGJIGSAW, REDROVER is a simple game with the potential to be extended to real world scenarios. Examples include the exploration of an unknown area by a group of people or robots. If enhanced with some kind of mapping mechanism, the interaction pattern of REDROVER could easily enable users to acquire the elements of a region and share these results as they meet other users.

Design and Implementation. A key issue in REDROVER is to disseminate information about the physical location of a given user. In our implementation, this information is encoded as a tuple. How tuples are filled with location information, and hence the back-end of the location support, is decoupled from the rest of the

application through an appropriate API, and thus can easily accommodate various positioning alternatives. Currently, we support feeding of location data from a GPS device.

Instead of "pushing" the location information to other users, in our design we let each application agent "pull" it from the tuple space of the other users. At start-up, all application agents register a weak ONCEPERTUPLE ubiquitous reaction, whose pattern matches any tuple containing location information. Each time a player moves, its location is updated by writing a tuple containing the corresponding information in the tuple space. This causes the reaction listener of all application agents (including the one corresponding to the moving user) to fire, and update the display according to the new location.

Detection of player disconnection relies on the LimeSystem tuple space and its support for reactions. In this tuple space, tuples corresponding to hosts and agents contain a field reporting the current connection status of the mobile unit. In RED-ROVER, all agents are registered for tuples in the LimeSystem representing departed hosts: the corresponding listener updates the display by substituting the icon of the player with the ghost icon. Similarly, when a player reconnects a complementary reaction fires that changes her icon back to normal. Interestingly, thanks to the ONCEPERTUPLE reaction associated to location tuples, no further action is needed to retrieve the current location of the reconnected user. If the player did not change its position while disconnected, its location tuple will still be the same and no reaction will fire. On the other hand, if the location tuple is different at engagement time, then the corresponding listener will execute, updating the display.

To enforce a separation between team data and location information, we exploited LIME's ability to define tuple spaces with different names, and hence are shared independently. Therefore, general information such as player location is written to a game-wide tuple space, while team-specific information like flags, or images is written to a tuple space accessed only by the members of that team. A player can either request to be notified when a given piece of information appears in its team tuple space, or can probe a specific player for a specific object. For instance, flag capture notification is implemented with a ONCE weak reaction over the team federated tuple space. Instead, the feature allowing retrieval of the player's icon is implemented through a **rdp** that goes directly to the tuple space of the corresponding remote player.

6. DISCUSSION

In this section we elaborate on the presentation thus far. First, we report about lessons learned from the design and development of LIME. Then, we broaden the scope of our discussion and report about other projects that either build on or are inspired by LIME. Finally, we turn our attention to efforts that are not directly related with LIME and survey related research projects that deal with tuple spaces and middleware for mobility.

6.1 Reflections and Lessons Learned

LIME is the result of a continuous interplay among the definition of the underlying formal model, the design and implementation of the middleware, and its evaluation with mobile applications. The development of a model for LIME and its

formalization favored a better understanding of the abstractions provided by the middleware. In particular, by keeping the programming interface as close as possible to the operations defined in the formal model, we made it easy to communicate and reason about the functionality of the system and its use in applications. In an incidental way, this task also provided an evaluation of the applicability of Mobile UNITY to the specification of a middleware for mobility. The ability to think about abstractions in a setting unconstrained by implementation details favored a style of investigation characterized by a more radical perspective, where the decisions driving the modeling and the definition of the main abstractions were mostly determined by the need for expressiveness and completeness.

Building applications on top of LIME made it possible for us to evaluate the usefulness of its programming abstractions and constructs. For example, our experience confirmed the effectiveness of both weak reactions on the federated tuple space and the ONCEPERTUPLE reaction mode to simplify programming. The registration of a single reaction is sufficient to guarantee notification of relevant data as it appears throughout the federation, independent of changes in configuration. Interestingly, this power has a cost: the implementation of ONCEPERTUPLE weak reactions is probably the most complicated portion of the current LIME software—this should be expected, since we are shifting a great deal of complexity away from the programmer and into the run-time support.

Another interesting by-product of these empirical evaluations is an understanding of the programming and architectural styles fostered by LIME and recurring in mobile applications. One distinction can be made between applications such as ROAMINGJIGSAW whose main requirement is to enable sharing of data *despite* mobility and those such as REDROVER where most of the computation is driven by reactions to changes in context and whose functionality *exists because of* mobility. In these and other application typologies, a recurring dilemma is between an application style that provides a weakly consistent view of the system in the presence of mobility, and one that provides a fully consistent view that takes into account departure and arrival of mobile units. Choosing one representation style or the other has non-trivial implications on the complexity of the overall design and development task, as well as on the primitives that must be used. If weak consistency is enough, the view can be built incrementally by exploiting the data notification mechanism provided by weak reactions, usually in the ONCEPERTUPLE mode. If, instead, a fully consistent view is required, application-specific machinery must be written in addition to using the LimeSystem tuple space to react (immediately) to changes in the system configuration. In our experience both styles are naturally accommodated by the abstraction of a transiently shared tuple space and use of the LimeSystem tuple space. Our "developers", mostly graduate and undergraduate students, found it easy not only to *program* applications with LIME but, most importantly, to *think* about the application in terms of the metaphors characteristic of the underlying LIME model.

Despite the somewhat limited experience, analysis of LIME application developments revealed that the programming style induced by LIME is quite different from what we initially expected. This is especially true in the case of weak reactions and the LimeSystem tuple space. Because LIME was envisioned to be a coordina-

tion framework founded on the idea of transiently shared tuple spaces accessible exclusively through Linda operations, reactive programming was not even part of the initial core. Nevertheless, reactive programming, especially weak reactions on the federated tuple space, form critical pieces in both RedRover and RoamingJigsaw. Similarly, the LimeSystem was initially thought of as an add-on to support very specific needs. Actually, we initially thought that the explicit context knowledge provided by the LimeSystem tuple space could be bypassed by the observation of changes in the data context. Experience with our applications, especially RedRover, showed that this does not hold in general and that developers may use extensively the LimeSystem tuple space. On the other hand, RoamingJigsaw did not use the LimeSystem tuple space, showing clearly that it is not an essential part of all mobile application styles.

6.2 Lime-Inspired Works

Although this paper presents the first comprehensive description of Lime, the published model and implementation have already been influential in some research endeavors both inside and outside our research group.

Within our group, we have extended or exploited Lime for several purposes. First, we have begun to explore the issues of security in tuple space based ad hoc mobile environments [Handorean and Roman 2003] by allowing applications to protect selected tuple spaces and even individual tuples through the use of passwords. The same passwords were also used to encrypt communication among hosts when exchanging messages related to sharing specific tuples spaces. Second, we have used Lime as the foundation for a Jini-like service discovery mechanism [Handorean and Roman 2002]. This project, implemented as an application layer on top of Lime uses the tuple space for sharing service advertisements and performing pattern-based service discovery. This extends the client-server model of service discovery for the mobile ad hoc environment by coupling the services available for discovery with the services available in the network, and maintaining this connection even as connectivity changes. In another project the Lime tuple space is used to support code mobility by storing Java class bytecode [Picco and Buschini 2002]. The class loading mechanism is extended to resolve class names by searching the federated tuple space, instead of a well-known, centralized code repository. This mechanism enables the code on demand paradigm for code mobility in the mobile ad hoc environment, where connections to specific code servers are not always available. Finally, we have recently developed TinyLime [Curino et al. 2005], an extension of the Lime model designed for mobile data collection in wireless sensor networks. In TinyLime, applications running on the mobile base stations share the data they gather and obtain access to sensor data through the same transiently shared tuple space. An energy-aware implementation is provided for the Crossbow MICA2 platform.

Groups at other universities have presented alternatives to both the Lime implementation and the formalization. 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 [Carbunar et al. 2001]. On top of this restricted model, they proposed some ideas for tuple space security. A group from the University of Bologna proposed an alternative to the state-based formal specifi-

cation presented here, providing a calculus-based specification [Busi and Zavattaro 2001]. This calculus presents several alternatives to the original LIME model, 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 our development of a more abstract and general coordination concept and methodology called *Global Virtual Data Structures* (GVDS) [Picco et al. 2002]. 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 other research projects have picked up the GVDS theme and instantiated it in their own unique ways. The XMIDDLE [Mascolo et al. 2002] system developed at University College, 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 [Cugola and Picco 2002] 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.

6.3 Related Projects

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 [Freeman et al. 1999] and TSpaces [Lehman et al. 2001], respectively. These systems present a centralized tuple space, accessible through remote operations by multiple processes. In contrast, LIME provides a fully distributed, peer-to-peer implementation of the tuple space abstraction. Distributed Linda implementations have been studied extensively for fault tolerance [Xu and Liskov 1989; Bakken and Schlichting 1995] and data availability [Pinakis 1993]. 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 possible in the mobile environment. Instead, LIME supports application development in dynamic scenarios, namely those characterized by mobile ad hoc networks as well as mobile agents, and takes into account the fluidity of these environments both at the model and implementation level.

One of the first applications of Linda to mobility came in the Limbo platform [Blair et al. 1997; Wade 1999], 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 they migrate, no concept of reaction, and the mechanism for relocation of tuples is unclear. Interestingly, the Limbo *universal tuple space*, 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 [Omicini and Zambonelli 1999] and MARS [Cabri et al. 2000], 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 performed. While in LIME reactions form a core concept for the application programmer, MARS and TuCSoN reactions are designed to be implemented by *manager* agents only, and application agents use only the basic tuple space operations. These manager agents support *system* design 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. While both MARS and TuCSoN provide transparent access to the local tuple space, MARS adds an option to fully qualify a tuple space name for an operation, thus identifying the specific host where operation should be executed. This enables remote operation on tuple spaces, but connectivity must be available and the agent must be explicit about interaction. This is in contrast to the LIME model that performs operations transparently over the current context of transitively connected hosts. 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 built-in coordination or data exchange among tuple spaces such as LIME tuple migration.

The KLAIM [Nicola et al. 1998] 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, 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, 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 is found. No such guarantee can be given with an event model where the events are asynchronously delivered. Nonetheless, we also support this second approach through weak reactions.

In addition to the shared memory model, significant effort has focused on migrating other distributed computing models such as objects, events, and databases to the mobile environment. For example, Mobile CORBA [Adwankar 2001] supports a limited degree of mobility for clients accessing distributed objects. Unlike LIME, which targets the extreme MANET environment, Mobile CORBA works only in nomadic computing environments where mobile clients rely on a stable networking infrastructure. Event aggregation and dissemination are addressed in Solar [Chen and Kotz 2002], the Context Toolkit [Dey et al. 2001], and STEAM [Meier and Cahill 2002], while Bayou [Terry et al. 1995] provides a replicated database model for the MANET environment. In some respects, LIME combines ideas from research on both data-centric and event-centric models under a single unified framework, where the novel idea of transiently sharing the data repositories is combined with the enhanced expressive power of state-based reactions.

Apart from models, research has focused also on specific problems that are important for mobile applications, such as disconnected operation, replication, and adaptation. Coda [Kistler and Satyanarayanan 1992] is one of the first systems to address disconnected operation, specifically supporting user-defined conflict resolution of file modifications. Replication [Boulkenafed and Issarny 2003], consistency, and availability of information have also been addressed, using user-defined profiles to guide the system replication. Puppeteer [deLara et al. 2001] specifically deals with adaptation of data, e.g., scaling of images, for remote, wireless access, while MIDAS [Popovici et al. 2003] takes an aspect-oriented approach for the environment to push context-dependent updates to applications. While none of these directions are the core focus of LIME, some of these ideas can be supported as services on top of LIME, using the shared tuple space model to exchange application, contextual, and control information. For example, we have begun to explore a replication layer that locally copies tuples according to user-specified patterns and consistency models [Murphy and Picco 2006].

Finally, several large endeavors have recently emerged to build middleware systems to support ubiquitous computing. For example, Gaia [Roman et al. 2002] provides a distributed operating system to support *active spaces*, offering a range of services including access to location information, resource management, task management, and event distribution. Aura [Sousa and Garlan 2002] provides similar features through a distributed architecture whose goal is to allow computation devices to disappear into the environment. Although these systems and LIME can both be classified as middleware, their respective focus is very different. Gaia, Aura, and similar efforts focus on providing a comprehensive service platform using standard communication facilities and abstractions (e.g., sockets and RPC), and target primarily nomadic computing scenarios where mobile nodes rely on a fixed infrastructure. Instead, LIME complements these efforts by placing itself at

the level of the core coordination facilities, where it contributes novel abstractions meant to simplify the development of higher-level services and applications, and does so by targeting the more complex mobile ad hoc networking environment.

7. 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 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. 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.

LIME continues to be developed as an open source project, available under the GNU 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

ADWANKAR, S. 2001. Mobile CORBA. In *3rd International Symposium on Distributed Objects and Applications (DOA)*. 52–63.

BACK, R. AND SERE, K. 1990. Stepwise refinement of parallel algorithms. *Science of Computer Programming 13,* 2-3 (May), 133–180.

BAKKEN, D. AND SCHLICHTING, R. 1995. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems 6,* 3 (March), 287–302.

BLAIR, G., DAVIES, N., FRIDAY, A., AND WADE, S. 1997. Quality of service support in a mobile environment: An approach based on tuple spaces. In *Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQoS'97)*. New York, USA, 37–48.

BOULKENAFED, M. AND ISSARNY, V. 2003. A middleware service for mobile ad hoc data sharing, enhancing data availability. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*. Rio de Janeiro, Brazil, 493–511.

BUSI, N. AND ZAVATTARO, G. 2001. Some thoughts on transiently shared dataspaces. In *Proceedings of the Workshop on Software Engineering and Mobility, co-located with the 23rd International Conference on Software Engineering*. Toronto, ON, Canada, 328–333.

CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. 2000. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing 4,* 4 (July-August), 26–35.

CARBUNAR, B., VALENTE, M., AND VITEK, J. 2001. LIME revisited: Reverse engineering an agent communication model. In *Proceedings of the 5th International Conference on Mobile Agents*. Atlanta, GA, USA, 54–69.

CARRIERO, N., GELERNTER, D., AND ZUCK, L. 1995. Bauhaus Linda. In *Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*. Springer-Verlag, London, UK, 66–76.

CHANDY, K. AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.

CHEN, G. AND KOTZ, D. 2002. Solar: An open platform for context-aware mobile applications. In *Proceedings of the 1st International Conference on Pervasive Computing*. Zurich, Switzerland, 41–47.

CUGOLA, G. AND PICCO, G. P. 2002. Peer-to-peer for Collaborative Applications. In *Proceedings of the International Workshop on Mobile Teamwork Support, co-located with the 22nd International Conference on Distributed Computing Systems*. IEEE Press, Vienna (Austria), 359–364.

CURINO, C., GIANI, M., GIORGETTA, M., GIUSTI, A., MURPHY, A. L., AND PICCO, G. P. 2005. Mobile data collection in sensor networks: The TinyLIME middleware. *Elsevier Pervasive and Mobile Computing Journal 4,* 1 (December), 446–469.

DELARA, E., WALLACH, D., AND ZWAENEPOEL, W. 2001. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*. San Francisco, CA, USA, 159–170.

DEY, A., SALBER, D., AND ABOWD, G. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction (HCI) Journal—Special Issue on Context-Aware Computing 16,* 204, 97–166.

FREEMAN, E., HUPFER, S., AND ARNOLD, K. 1999. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education.

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

HANDOREAN, R. AND ROMAN, G.-C. 2002. Service provision in ad hoc networks. In *Proceedings of the 5th International Conference on Coordination Models and Languages*, F. Arbab and C. Talcott, Eds. LNCS 2315. Springer, York, UK, 207–219.

HANDOREAN, R. AND ROMAN, G.-C. 2003. Secure sharing of tuple spaces in ad hoc settings. *Electronic Notes in Theoretical Computer Science 85,* 3, 1–20.

KISTLER, J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems 10,* 1, 3–25.

LEHMAN, T., COZZI, A., XIONG, Y., GOTTSCHALK, J., VASUDEVAN, V., LANDIS, S., DAVIS, P., KHAVAR, B., AND BOWMAN, P. 2001. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks 35,* 4 (Mar.), 457–472.

MASCOLO, C., CAPRA, L., ZACHARIADIS, S., AND EMMERICH, W. 2002. XMIDDLE: A data-sharing middleware for mobile computing. *Kluwer Personal and Wireless Communications Journal 21,* 1 (April), 77–103.

MCCANN, P. AND ROMAN, G.-C. 1998. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering 24,* 2, 97–110.

MCCANN, P. AND ROMAN, G.-C. 1999. Modeling Mobile IP in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology 8,* 2 (April), 115–146.

MEIER, R. AND CAHILL, V. 2002. STEAM: Event-Based middleware for wireless ad hoc networks. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems*. 639–644.

MURPHY, A. AND PICCO, G. P. 2006. Using LIME to Support Replication for Availability in Mobile Ad Hoc Networks. In *Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION 2006)*. Number 4038 in Lecture Notes on Computer Science. Springer, Bologna (Italy). To appear.

MURPHY, A. L., PICCO, G. P., AND ROMAN, G.-C. 2001. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing*

Systems (ICDCS-21), F. Golshani, P. Dasgupta, and W. Zhao, Eds. Phoenix (Mesa), Arizona, USA, 524–533.

Nicola, R. D., Ferrari, G., and Pugliese, R. 1998. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering 24,* 5, 315–330.

Omicini, A. and Zambonelli, F. 1999. Tuple centres for the coordination of internet agents. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. San Antonio, TX, USA, 183–190.

Picco, G. P. 1998. μcode: A lightweight and flexible mobile code toolkit. In *Proceedings of the 2^{nd} International Workshop on Mobile Agents*. LNCS 1477. Springer, 160–171.

Picco, G. P. and Buschini, M. 2002. Exploiting transiently shared tuple spaces for location transparent code mobility. In *Proceedings of the 5^{th} International Conference on Coordination Models and Languages*, F. Arbab and C. Talcott, Eds. LNCS 2315. Springer, York, UK, 258–273.

Picco, G. P., Murphy, A. L., and Roman, G.-C. 1999. Lime: Linda meets mobility. In *Proceedings of the 21^{st} International Conference on Software Engineering*, D. Garlan, Ed. 368–377.

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

Picco, G. P., Roman, G.-C., and McCann, P. 2001. Reasoning about code mobility with Mobile Unity. *ACM Transactions on Software Engineering and Methodology 10,* 3 (July), 338–395.

Pinakis, J. 1993. Using linda as the basis of an operating system microkernel. Ph.D. thesis, University of Western Australia, Australia.

Popovici, A., Frei, A., and G.Alonso. 2003. A proactive middleware platform for mobile computing. In *Proceedings of the 4^{th} ACM/IFIP/USENIX International Middleware Conference*. Rio de Janeiro, Brazil, 455–473.

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

Roman, G.-C., McCann, P., and Plun, J. 1997. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology 6,* 3, 250–282.

Roman, G.-C., Murphy, A. L., and Picco, G. P. 2000. Coordination and Mobility. In *Coordination of Internet Agents: Models, Technologies, and Applications*, A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, Eds. Springer, 254–273.

Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., and Nahrstedt, K. 2002. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 74–83.

Rosenblum, D. and Wolf, A. L. 1997. A design framework for internet-scale event observation and notification. In *Proceedings of the 6^{th} European Software Engineering Conference held jointly with the 5^{th} ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'97)*. Number 1301 in LNCS. Springer, Zurich (Switzerland), 344–360.

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

Sousa, J. and Garlan, D. 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, J. Bosch, M. Gentleman, C. Hofmeister, and J. Kuusela, Eds. Kluwer Academic Publishers, 29–43.

Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Operating Systems Review 29,* 5, 172–183.

Wade, S. 1999. An investigation into the use of the tuple space paradigm in mobile computing environments. Ph.D. thesis, Lancaster University, England.

Xu, A. and Liskov, B. 1989. A design for a fault-tolerant, distributed implementation of Linda. In *Digest of Papers of the 19^{th} International Symposium on Fault-Tolerant Computing*. 199–206.

Received ????; ????; accepted February 2006