LIME: Linda Meets Mobility

Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman
Dept. of Computer Science, Washington University
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
+1-314-935-{7536,7537,6132}
{picco,alm,roman}@cs.wustl.edu

Abstract
LIME is a system designed to assist in the rapid development of dependable mobile applications over both wired and ad hoc networks. Mobile agents reside on mobile hosts and all communication takes place via transiently shared tuple spaces distributed across the mobile hosts. The decoupled style of computing characterizing the Linda model is extended to the mobile environment. At the application level, both agents and hosts perceive movement as a sudden change of context. The set of tuples accessible by a particular agent residing on a given host is altered transparently in response to changes in the connectivity pattern among the mobile hosts. In this paper we present the key design concepts behind the LIME system.

1 INTRODUCTION

Today's users demand ubiquitous network access independent of their physical location. This style of computation, often referred to as *mobile computing*, is enabled by rapid advances in the wireless communication technology. The networking scenarios enabled by mobile computing range roughly between two extremes. At one end, the availability of a fixed network is assumed, and its facilities are exploited by the mobile infrastructure—this is the case with Mobile IP [10]. At the other end, the fixed network is absent and all the network facilities (e.g., routing) must be implemented by relying only on the available mobile hosts—this is the assumption made by research on *ad hoc* networks [6]. The characteristics of the wireless communication media, e.g., limited bandwidth and frequent disconnection, favor a decoupled and opportunistic style of computation. Computation is decoupled in that it is expected to proceed even in the presence of disconnection—a frequent event in this domain. Computation is opportunistic in that it exploits connectivity whenever it becomes available. This peculiar style of computation demands a new approach to the crafting of distributed applications.

Alongside this form of *physical mobility*, in which the hosts roam across the physical space and modify the topology of the network, researchers are paying increasing attention to various forms of *logical mobility* as well. In the latter case, often referred to as *code mobility* [2], the entity being migrated is the code running on the hosts, which are usually not allowed to move. Code relocation is expected to offer several advantages over the traditional client-server paradigm, the most important being enhanced flexibility, customizability, and reduced network traffic. Flexibility and customizability are provided by the ability to send to the server a portion of the code available to the client, or vice versa. The opportunity for a reduction in network traffic results from the possibility to minimize the number of interactions that take place across the network. In this paper we present a system called LIME (Linda in a Mobile Environment) which provides application designers and implementors with a minimalist set of constructs to deal with both physical and logical mobility. In our system, physical mobility involves the movement of mobile hosts, such as laptops or PDAs. Logical mobility is concerned with the movement of *mobile agents*, i.e., processes that are able to migrate from one host to another while preserving their code and state. The underlying assumption in our work is that both mobile hosts and mobile agents can be regarded as instances of a generic concept of mobile component, and *coordination* takes place through the use of transiently shared *tuple spaces* accessed via the basic set of Linda primitives [4].

In Linda, coordination is achieved through a tuple space globally shared among components which, independent of their actual location, can access the tuple space by inserting, reading, or withdrawing tuples containing information. The model provides both spatial and temporal decoupling. The components do not need to co-exist in time for them to communicate and can reside anywhere in the distributed system. Since decoupling is intrinsic to mobility, the Linda model is a natural choice for our system.

LIME retains the basic philosophy and goals of Linda while adapting them to mobility. Simple and rapid application development is facilitated by the same mechanisms which made parallel programming in Linda attractive to implementors. Programs (written in a variety of languages) view the world as a sea of tuples accessible by contents. Movement, logical or physical, results in implicit changes of the tuple space accessible to the individual components. The system, not the application program, is responsible for managing movement and the tuple space restructuring associated with connectivity changes. The formal semantic definition of LIME is the subject of a companion paper and relies on the use of Mobile UNITY [8], an extension of the UNITY notation and logic proposed by Chandy and Misra [1] with concepts that are fundamental in dealing with mobility.

The remainder of this paper is structured as follows. Section 2 provides a brief review of Linda. Section 3 motivates LIME and its design philosophy. Sections 4, 5, and 6 present a set of coordination primitives supporting transiently shared tuple spaces, location-aware computing, and reactive programming—the fundamental concepts underlying LIME. Section 7 highlights the technical challenges involved in implementing LIME and outlines several sample application scenarios. Finally, Section 8 summarizes our research contributions and highlights directions for future work.

2 LINDA

Linda has been proposed at the beginning of the past decade [4] as a new model of communication among concurrent processes. The fundamental abstraction provided to each process is a shared *tuple space* that acts as a repository of elementary data structures—the *tuples*. Each tuple is a list of typed parameters, such as ⟨"foo", 9, 27.5⟩, that contain the actual information being communicated. A tuple space is a multiset of tuples that can be accessed concurrently by several processes.

Tuples are added to a tuple space by performing an **out**(t) operation on it. After its execution, the tuple t is available to any subsequent operation on the tuple space. The update of the tuple space is performed atomically. Tuples can be removed from a tuple space by executing **in**(p). Tuples are anonymous, thus their removal takes place through pattern matching on the tuple contents. The argument p is often called a *template*, and its fields contain either *actuals* or *formals*. Actuals are values; the parameters of the previous tuple are all actuals, while the last two parameters of ⟨"foo", ?integer, ?long⟩ are formals. Formals are 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. (Tuples can contain formals as well.) The matched tuple and the template must have the same arity. If multiple tuples match a template, the one returned by **in** is selected non-deterministically and without being subject to any fairness constraint. The **in** operation is blocking, i.e., if no matching tuple is available in the tuple space the process performing the **in** is suspended until a matching tuple becomes available. Tuples can also be read from the tuple space using the **rd** operation. The execution of a **rd**(p) proceeds identically to **in**, except for the fact that the tuple matched and delivered to the process that executes the operation is copied rather than withdrawn from the tuple space. Similarly to the **in** operation, **rd** is blocking. Linda implementations typically include also an **eval** operation which provides dynamic process creation and enables deferred evaluation of tuple fields. For our purposes, we use hereafter only **out**, **in**, and **rd**.

Communication in Linda is decoupled in *time* and *space*. Decoupling in time refers to the fact that senders and receivers do not need to be in communication in order to exchange information. Tuples are stored in the tuple space and can be retrieved later, even if the process that produced the tuple terminated its execution already. Decoupling in space refers to the fact that a tuple in the tuple space is available to processes dispersed on the nodes of a distributed system—the actual location of a tuple is hidden from the tuple producer and consumer. Decoupling is appealing, as it separates clearly the behavior of the individual processes from the communication needed to coordinate their actions. The idea enjoys wide acceptance in many scientific communities ranging from economics to artificial intelligence, and is at the core of a new interdisciplinary research area that investigates technologies and methodologies for the *coordination* of components in complex systems [7, 3]. Notably, the decoupling between components and their coordination fostered by Linda has several points of contact with the distinction between components and their interconnection that constitutes the core of recent research on software architecture [11]. We chose the Linda model as the basis for this work due to its minimality and decoupling in time and space.

3 LINDA EXTENSIONS FOR A MOBILE ENVIRONMENT

Linda provides coordination among concurrently executing components accessing a shared tuple space that is persistent, globally accessible, and statically created. Maintaining these properties in the presence of physical mobility is complicated, because connectivity can no longer be taken for granted. Early research on fault-tolerant distributed implementations of Linda [12] tackled the problem under the assumption that disconnection was just an unfortunate accident, and employed replication schemes to increase tuple availability. However, in physical mobility disconnection is often forced explicitly by the user, e.g., to save battery power during movement or to reduce communication costs over expensive cellular phone lines. To make matters worse, mobile hosts (and mobile agents) are completely independent and controlled by users occasionally forming transient communities; thus, they may come in contact once and never again. This makes unreasonable any assumptions about eventual delivery of data. Finally, logical mobility complicates the implementation of a distributed tuple space even in the absence of any physical mobility. For instance, replication schemes that rely on the locality of processes may no longer be applicable, because processes can move freely around the network. The idea of a static, persistent, and globally visible tuple space is then unreasonable. Mobility demands weaker constraints on the tuple space and dynamic reconfiguration of its contents.

In the model underlying LIME, mobile agents are programs that can travel among mobile hosts. They are "active" components of the system. Mobile hosts are roaming containers for agents, to which they provide connectivity. The Linda model is adapted by LIME through the notion of a *transiently shared tuple space* that ties together physical and logical mobility. Tuple spaces are permanently bound to mobile agents and mobile hosts. Transient sharing enables dynamic reconfiguration of their contents according to agent migration or connectivity variations. At a high level of abstraction, mobile agents interacting in the logical space provided by a single host are akin to mobile hosts interacting in the physical space spanned by communication links. Hence, transiently shared tuple spaces and the associated access primitives provide a single coordination toolkit for both scenarios. In practice, however, distribution and mobility do complicate implementability. As a result a more constrained use of the LIME primitives is allowed when physical mobility is present.

4 TRANSIENTLY SHARED TUPLE SPACES

The fundamental abstraction provided by LIME is the notion of a *transiently shared tuple space*. As summarized in Figure 1, in LIME each mobile agent has access to an *interface tuple space* (ITS) that is permanently associated with that agent and transferred along with it when movement occurs. Each ITS contains information that the mobile agent is willing to share with others, and is accessed using the conventional Linda operations **in**, **rd**, and **out** described in Section 2, whose semantics are unaffected. On the other hand, the actual content of the ITS is determined differently from Linda. The set of

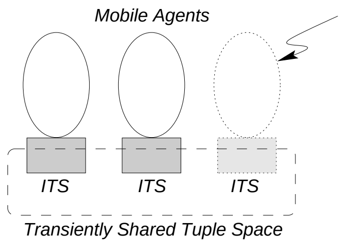

Figure 1: Transiently shared tuple spaces in LIME.

tuples that can be accessed through the ITS is dynamically recomputed in such a way that, for each mobile agent, the content of its ITS gives the appearance of having been merged with those of the other mobile agents which are currently co-located. This way, each mobile agent "sees" through its own ITS the same transiently shared tuple space presented to the others. Operations performed on the ITS are effectively performed on the contents of the transiently shared tuple space; e.g., if agents A and B are co-located and A performs an **out**(t) on its ITS, after the tuple t is inserted in A's ITS it is available for retrieval with an **in**(t) by agent B.

The tuple space that can be accessed through the ITS of an agent is *shared* by construction and is *transient* because its content changes according to the migration of agents. The action of making the contents of an ITS accessible to other agents through the transiently shared tuple space takes place in reaction to changes in the set of co-located agents. Upon arrival of a new mobile agent, the transiently shared tuple space is recomputed by taking into account the ITS of the new agent. The result is made accessible to all the agents currently co-located. This sequence of operations is called *engagement* of the tuple spaces, and is performed as a single atomic transaction. Similar considerations hold for the departure of a mobile agent, resulting in the *disengagement* of the corresponding ITS. Its content is removed atomically from the transiently shared tuple space according to rules that are discussed later.

In LIME, agents may have multiple ITSs, and also *private* tuple spaces, i.e., not subject to sharing. Tuple spaces are *named*; the name effectively defines a notion of typing for the tuple space and, in the case of ITSs, determines the sharing rule. If an agent has multiple ITSs, these are shared independently with the corresponding ITSs of other co-located agents, if any. In other words, if agent A owns the ITSs named S, T, and U, while agent B owns the ITSs named T, U, and V, only T and U will become transiently shared between A and B. For instance, this enables an agent to exchange information with a service broker about the available CD resellers by transiently sharing the corresponding ITS, and then subsequently

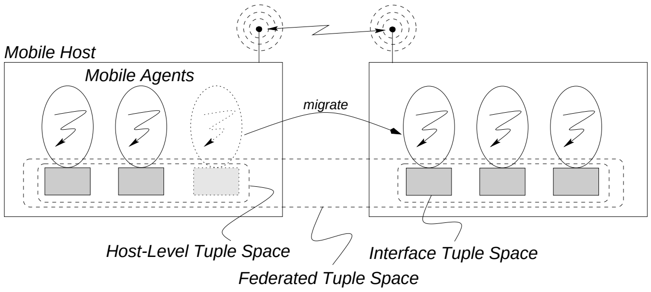

Figure 2: Transiently shared tuple spaces to handle physical and logical mobility.

share information about a given title and the payment options with the reseller selected through a different ITS, thus keeping separate the information concerned with different tasks and different roles. By construction, all agents are bound to a *LimeSystem* ITS whose tuples can be read but not withdrawn. This ITS contains system information concerning the agent, e.g., its identifier, as well as information concerned with the host, e.g., quality of service information. Transient sharing of *LimeSystem* enables co-located agents to access global system information. We will detail the role of this tuple space later on. To identify the tuple space on which a given operation is performed, we use the dot notation, e.g., $T.\textbf{out}(t)$. However, in this paper we will focus on agents with a single ITS, unless otherwise specified, and drop the name of the tuple space.

So far, the discussion has been focused on mobile agents. However, LIME applies the notion of transiently shared tuple space to a generic mobile component regardless of its nature—physical or logical. This relies on an extended notion of connectivity that encompasses both kinds of components. Mobile hosts are connected if a communication link connecting them is *available*. Availability may depend on a variety of factors, including quality of service, security considerations, or connection cost; in principle, all of these can be represented in LIME. However, in this paper we limit ourselves to a simple notion of availability based on the presence of a functioning link. Because we assume bidirectional links and the presence of routing capabilities in the physical network, our notion of connectivity is commutative and transitive. Mobile agents are connected if 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 link. 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. Finally, we assume that both mobile agents and mobile hosts are given globally unique identifiers. Figure 2 depicts the model adopted by LIME.

The content of the ITS of each mobile agent is determined by the presence of connectivity, in the aforementioned extended meaning. By definition, agents co-located on the same host are connected, and this creates a *host-level tuple space* that is transiently shared among all such agents and accessible through each agent's ITS. As evident in Figure 2, the host-level tuple space can be regarded as the ITS of a mobile host, as it is permanently associated with it; if no mobile agents are currently hosted, the host-level tuple space is empty. Next, the mechanism of transient sharing is applied to the host-level tuple spaces. Hosts that are connected merge their host-level tuple spaces into a *federated tuple space* whose content is transiently shared across the network. A tuple in the ITS of an agent can be either local and thus belonging to the local host-level tuple space, or remote and thus belonging to the host-level tuple space of a mobile host that is currently accessible.

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, there is no place to store a persistent tuple space. Connection among machines comes and goes and the tuple space must be partitioned in some way. In the scenario of logical mobility, maintaining locality of tuples with respect to the agent they belong to may be complicated. LIME enforces an a priori partitioning of the tuple space in subspaces that get transiently shared according to precise rules, providing a tuple space abstraction that depends on connectivity. In a sense, LIME takes the notion of decoupling proposed by Linda further, by effectively decoupling the mobile components from the global tuple space used for coordination.

In this model physical and logical mobility are separated in two different tiers of abstraction. It is worth noting, however, that 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 a host-level tuple space as a local communication mechanism among co-located agents.

5 LOCATION-AWARE COMPUTING

The transient nature of the LIME shared tuple space poses some additional challenges, as illustrated in Fig-

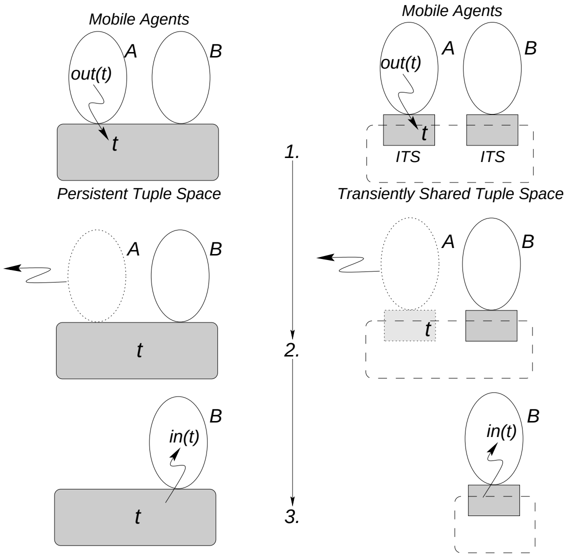

Figure 3: Persistent vs. transiently shared tuple spaces.

ure 3. Let us consider a simple system composed of two mobile agents, A and B. A is initially co-located with B; before departing, A leaves some information that is intended to be eventually processed by B. This simple scenario, depicted in the left hand side of Figure 3, is represented straightforwardly in Linda. A performs **out**(t) and deposits in the tuple space the tuple t containing the information to be communicated to B (step 1). Thanks to the persistence and global availability of the tuple space, the tuple t remains in the tuple space even after A departs (step 2), thus it is still available when B eventually tries to withdraw it by performing **in**(t) (step 3). The situation with transiently shared tuple spaces is depicted on the right hand side of the same figure. Since A and B are initially co-located, when A performs **out**(t) the tuple t is inserted in the ITS of A and becomes available to B thanks to the transient sharing of the agents' ITSs (step 1). However, when A departs, it takes along its own ITS (step 2). If B tries to pick up the tuple t after A has already departed, the corresponding **in**(t) will be performed on a transiently shared tuple space consisting of B's ITS only, and thus may block since t is no longer present there and there might not be any other tuple satisfying the match (step 3).

This problem is a consequence of the fact that in LIME there is no well-known persistent tuple space that can be used as a global repository of tuples. The configuration of the tuple space, i.e., its content, varies dynamically according to the location of components. Still, it is desirable to retain the advantage provided by the decoupling in time and space characterizing the Linda model of communication. In our example, from B's perspective it would be desirable for the withdrawal of t to be possible any time after A becomes co-located with B. To accomplish this, A should be allowed to specify that the effect of an **out**(t) on the transiently shared tuple is to place t in B's ITS rather than keeping it in A's ITS as we assumed so far.

In LIME this is accomplished by exploiting the notion of *location*—central to mobility in general. The location of a tuple is a tuple space. In the case of an ITS, the location of a tuple is identified uniquely by the name of the tuple space and by the identifier of the mobile agent owning the ITS, since the agent and its ITS are permanently bound. Given this notion, in LIME a tuple can be placed into the ITS T of a mobile agent λ by simply using T.**out**[λ](t), a version of **out** annotated with the tuple's intended location. The semantics of the **out**[λ] operation take into account the location of agents, and involve conceptually two steps. The first step is equivalent to a conventional **out**(t), the tuple t is inserted in the ITS of the agent calling the operation, say ω. At this point the tuple t has a *current location* ω, and a *destination location* λ. If the agent λ is currently connected, i.e., either co-located or located on a connected mobile host, the LIME system reacts to the **out** operation by moving the tuple t to the destination location. The combination of the two actions—the insertion of the tuple in the ITS of ω and its instantaneous migration to the ITS of λ—are performed as a single atomic operation. On the other hand, if λ is not currently connected, the tuple remains at the current location, the ITS of ω.

Thus, in our example, A could circumvent the problem described earlier by performing **out**[B](t) on its ITS. Note that performing **out**[λ](t) does not necessarily imply guaranteed delivery of t to λ; the rules for nondeterministic selection of tuples as defined by Linda are still in place and the tuple t is always available in the transiently shared tuple space, even when it is not yet within the intended ITS. Thus, it might be the case that while waiting for λ to connect, or after it becomes connected and t is transferred to λ's ITS, some other agent may withdraw t from the tuple space before λ.

User-specified tuples are automatically augmented by the run-time support with fields recording their current and destination locations. This information enables LIME to detect the presence of "misplaced" tuples (i.e., tuples whose current location is different from their intended destination) and facilitates state reconciliation upon engagement and disengagement of tuple spaces. For instance, if t still belongs to the ITS of ω, and λ becomes connected, the system detects the presence of the misplaced tuple t and migrates it to the ITS of λ, also changing the current location of t to the value of its destination, λ. Since this action is part of the engagement of tuple spaces, the actions of becoming connected,

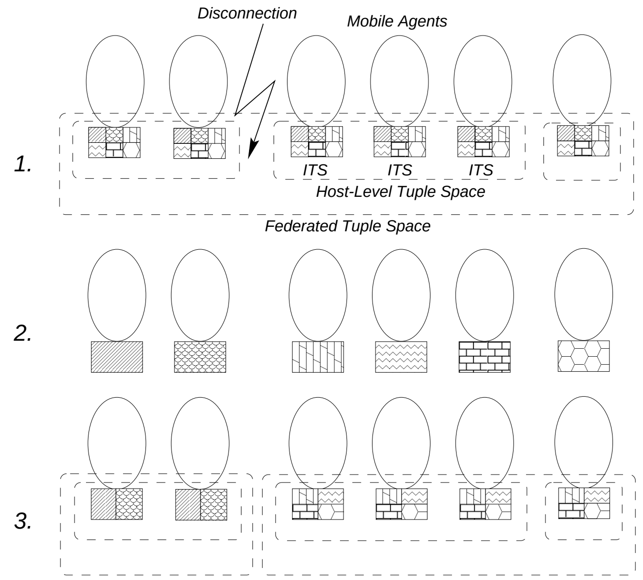

Figure 4: Recomputing transiently shared tuple spaces on disengagement.

merging of the ITS, and migrating misplaced tuples take place in a strictly sequential order and are executed as if they were a single atomic operation.

Disengagement relies on the notion of location, as well. When an agent departs, the transiently shared tuple space that is presented to mobile agents must be recomputed accordingly. Conceptually, this process can be described as depicted in Figure 4. Initially (step 1), the ITS of each mobile agent has access to the (bag) union of the tuples contained in the physical ITS actually associated with the agent. This is represented by using different fill patterns for the boxes representing the ITSs. Upon occurrence of a disconnection, be it the departure of an agent or a physical disconnection like in Figure 4, the transiently shared tuple spaces are partitioned into their constituents (step 2) such that the ITS of each agent, say ω, contains only the portion of the shared tuple space it is responsible for, i.e., all the tuples whose current location is ω. This includes also misplaced tuples, whose final destination is different from ω; if the destination is not present, placing these tuple with the agent that generated them is the only meaningful solution. Thus, this step basically partitions the contents of the transient tuple spaces back into the "concrete" ITSs that are actually physically carried by each mobile agent. Finally, these ITS are merged again according to the new configuration of the system, thus completing the tuple space update process caused by disconnection (step 3).

The ability to abstract away the location of mobile agents and hosts as well as to access their data seamlessly simplifies certain programming tasks. However, there are situations where it may be desirable to limit access only to a portion of a transiently shared tuple space, either for programming convenience or for performance reasons. For instance, the programmer may be temporarily concerned only with the ITS of a given agent or a particular host-level tuple space. These situations demand new forms of **in** and **rd** annotated with locations, similar to **out**$[\lambda]$. In LIME, annotations for these operations come in the form **in**$[\omega, \lambda]$ and **rd**$[\omega, \lambda]$, where the current and destination locations defined earlier are used. Either of the two locations can be left unspecified, in which case the symbol "_" will be used. Different combinations of location parameters identify different projections of the transiently shared tuple space. In the following, we review first the combinations allowed in LIME when dealing with logical mobility on a single host. Later on, we show what operations are permitted in the presence of physical mobility. The discussion focuses on the **in** operation.

The operation **in**$[\omega, \_]$ performs an **in** on the projection of the transient tuple space whose tuples have current location equal to ω. This operation provides a means to restrict the scope of the **in** operation to the ITS of a single agent, including misplaced tuples that have been created by this agent. LIME allows one also to refer to tuples that are in a given ITS but whose destination is actually another agent λ, in the form **in**$[\omega, \lambda]$. The ability to perform these operations can be useful, for instance, in a kind of tuple "garbage collection." If a garbage collector agent has some application knowledge that it will never meet the agent λ again, then it can purge useless tuples from the ITS of ω. As a special case, **in**$[\omega, \omega]$ performs the **in** on the tuples in ω's ITS that are not misplaced, i.e., whose destination is ω and whose current location is ω as well. **in**$[\_, \lambda]$ restricts the scope of the **in** to the tuples in the transiently shared tuple space whose final destination is λ. If λ is currently not present, such tuples are currently misplaced in the ITS of some other agent. If λ is present, then the operation becomes equivalent to **in**$[\lambda, \lambda]$ because, according to semantics of transient sharing, if λ is present all misplaced tuples directed to it migrate atomically to the correct ITS at the time of engagement.

When physical mobility is present, LIME provides also the capability to restrict the query performed by **in** to a specific host-level tuple space. This is achieved by specifying the identifier Ω of a mobile host as the first parameter, e.g., **in**$[\Omega, \_]$. In general, the current location can be specified either as the identifier of a mobile agent or of a mobile host. On the other hand, the destination location must always be the identifier of a mobile agent. This restriction holds also for the **out**$[\lambda]$ operation whose λ parameter, representing the destination location for t, cannot be a mobile host. If this were allowed, the tuple would eventually belong to a host-level tuple space. However, we already discussed that this tuple space is actually made of the concrete ITSs associated to mobile agents, they are the ones physically holding tuples. According to our model, if no agent is specified as a destination location for t, the tuple vanishes upon disconnection. Nevertheless, if in some applications the host must own a tuple space, this can be realized in LIME by using a stationary and persistent agent that is responsible for holding incoming tuples destined to that host and for making them available to the agents located there.

6 REACTIVE PROGRAMMING

In the rapidly changing environment that characterizes mobility, the ability to react to events, and to do it as soon as possible, is of great importance. Events can be concerned with the physical environment, like disconnection or changes in the quality of service, or with the application, like the availability of data carried by other parties. A typical example might be the arrival or departure of a mobile agent.

Events are naturally represented in the Linda model as tuples. This is actually the solution exploited by LIME as well. It leverages off the *LimeSystem* ITS described earlier. The run-time support, which can be modeled as a stationary agent, continuously monitors the underlying layers of the virtual machine for system events. These events, that include departure or arrival of mobile agents, changes in connectivity, and changes in the quality of service, are transformed into event tuples and are inserted in the *LimeSystem* ITS of the stationary agent. There, this information becomes available to all the agents connected through the federated tuple space generated by transient sharing of all the *LimeSystem* ITSs.

In Linda, the **in** operation already provides a basic mechanism for coping with events. Processes can suspend on the occurrence of an event, modeled by the appearance of the corresponding tuple in the tuple space, and will take appropriate actions after the tuple is retrieved by the **in**. However, this solution has some drawbacks. From a semantic point of view, there is no guarantee that the event is actually caught by a process λ interested in it. Another process μ could be interested in the same event and perform the **in** operation before λ. To guarantee event delivery, more complex schemes must be used. They must involve a priori knowledge about the number of processes interested in an event. From a performance point of view, since **in** is blocking, listening to multiple events requires one thread of control per event, which is often impractical. The fundamental problem rests with the fact that Linda forces applications to "pull" out tuples, while reactive programming demands to have tuples "pushed" to applications. Finally, it is desirable to encapsulate the reaction to an event into its own definition, thus providing a higher level of abstraction to the programmer who should be freed from the burden of dealing explicitly with synchronization issues.

LIME introduces the notion of a *reactive statement* having the form T.**reactsTo**(s, p) where s is a code fragment containing non-reactive statements that is to be executed when a tuple matching the pattern p is found in the tuple space T. T can be any tuple space. However, in the next section, we will see that some additional constraints are necessary to deal with remote tuple spaces. The execution of **reactsTo** "registers" the reaction with T; a complementary operation to "de-register" the reaction is also provided by LIME. The semantics of the **reactsTo** statement are the same as those defined for Mobile UNITY reactive statements, described in [8]. After each non-reactive statement, a reaction is selected nondeterministically among those registered, and its guard is evaluated. If the guard is true, the corresponding action is executed, otherwise the reaction is equivalent to a skip. This selection and execution proceeds until there are no enabled reactions and the normal execution of the non-reactive statements can resume. Thus, reactive statements are executed as if they belong to a separate reactive program that is run to fixed point after each non-reactive statement. These semantics offer an adequate level of reactivity, because all the reactions registered are executed before any other statement of the co-located agents, including the migration statements. Thus, the programmer's effort in dealing with events is minimized.

Although in principle s and p could have arbitrary forms, in practice their structure is constrained by implementation considerations. For example, if the definition of p contained arbitrary expressions in the host language—Java in the current implementation of LIME—the truth of p would need to be evaluated after each statement. This would require either access to the innards of the Java virtual machine or the development of a higher level language built on top of Java whose run-time support manages arbitrary reactive statements. For this reason, in the current implementation of LIME the specification of p is reduced to a pattern that is matched against tuples in the tuple space on which the reactive statement is executed. This way, the only statement that can trigger a reaction is the insertion of a tuple in the tuple space, which is under the control of LIME. Similar issues relate to s. Conceptually, s could be any code fragment containing non-reactive statements, thus including tuple space operations. In practice, reactions are executed in a separate thread of control belonging to the LIME system. Thus, blocking operations are forbidden in s, as they would actually block the processing of all the reactions.

In general, the presence of tuple space operations in s complicates matters. From a performance point of view, s should be kept as small as possible to allow fast processing of reactions; as execution of tuple space operations is usually more demanding than conventional statements and they should be used with much care. From a semantic point of view, if an **out**(t) operation is allowed in s, t may match the pattern p specified by some other reaction and thus generate a potentially infinite reaction loop. LIME presently allows tuple space operations in s, although subject to the constraints described below. Our rationale stems from the notion that preventing the programmer from modifying reactively the tuple space is worse than leaving up to the programmer the evaluation of the semantic and performance tradeoffs. Application development with the LIME API will hopefully validate this hypothesis.

The actual form of the **reactsTo** operation is annotated with locations—this has been omitted so far to keep the discussion simpler. A reaction assumes the form T.**reactsTo**$[\omega, \lambda](s, p)$, where the location parameters have the same meaning as discussed for **in** and **rd**. However, reactive statements are not allowed on federated tuple spaces; in other words, the current location field must always be specified, although it can be the identifier either of a mobile agent or of a mobile host. The reason for this lies in the constraints introduced by the presence of physical mobility. If a federated tuple space is present, its content, accessed through the ITS of a mobile agent, actually depends on the content of ITSs belonging to remote agents. Thus, to maintain the requirements of atomicity and serialization imposed by reactive statements would require a distributed transaction encompassing several hosts for each tuple space operation on any ITS—very often, an impractical solution.

For these reasons, LIME offers the capability to react asynchronously to the availability of tuples in a remote ITS by providing an operation T.**upon**(s, p) that has weaker semantics than the **reactsTo**. Conceptually, it works as depicted in Figure 5, where an additional, private tuple space associated with each mobile agent is shown. The purpose of this *event tuple space* (ETS) is to collect event tuples and is introduced here only to simplify the presentation—it is not actually provided among LIME constructs. The semantics of **upon** can then be described as follows:

1. When **upon**(s, p) is invoked on the ITS T_λ of a given agent λ, a reaction **reactsTo**(s', p) is registered on each ITS currently shared with T_λ. A reaction **reactsTo**(s, p') is also registered on λ's ETS, E_λ.
2. When an **out**(t) operation with t matching the pattern p is performed on the ITS T_μ of a given agent μ, the reaction T_μ.**reactsTo**(s', p) fires and s' is executed. s' performs an E_μ.**out**(t') where t' is a copy of t augmented with information that enables the system to bind an event tuple to the agent that is performing the **upon**.
3. An asynchronous action moves the tuple from the E_μ to E_λ. There is no guarantee that this action happens as soon as t' shows up in E_μ, because this would require starting a distributed transaction and suspending the execution of all the connected agents. Instead, LIME guarantees eventual delivery of t' to E_λ, if connectivity is available.
4. When t' reaches E_λ, the reaction E_λ.**reactsTo**(s, t') fires, where s is actually the statement specified originally in the **upon**.

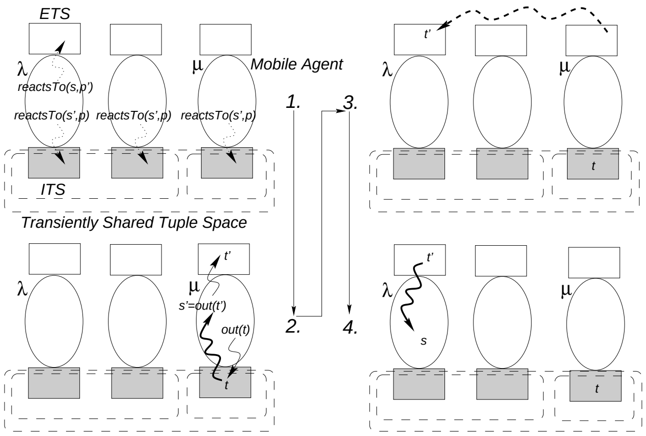

Figure 5: Reacting to remote events in LIME. Thick solid lines represent reactions, while the thick dotted line represents an asynchronous action.

The operation **upon** in LIME is similar to the `notify` operation provided by Sun's JavaSpace [9] and to the event registration mechanism provided by IBM's T-Spaces [5]. The development of a richer event model, allowing reactions to arbitrary events other than insertion of a new tuple, is the subject of on-going work.

7 DISCUSSION

The technical issues involved in the development of LIME are complex. For instance, in a fully mobile setting disconnection can take place at any time; tuple transfers may lead to distributed consensus problems and proper application level protocols are needed to prevent tuple duplication or loss. Matters are complicated further by the semantics of engagement, which requires the transfer of misplaced tuples to take place as if part of a distributed transaction. Tuple transfer is also affected by the availability of routing at the network layer, presently

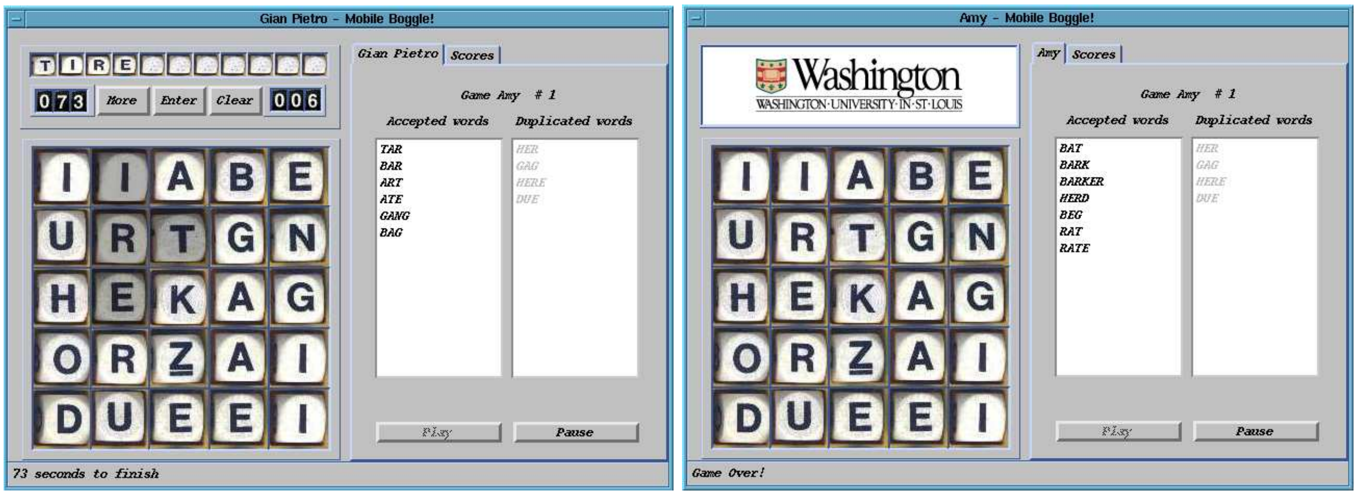

Figure 6: Mobile Boggle. The objective of Boggle^{TM} is to use adjacent letters on the board to form as many words as possible before the timer expires. Only words that are not found by others score points. In our version, a game is initiated by one player. Other players can join the game whenever connection is established with one of the already active players, and then disconnect. The game then proceeds in a disconnected fashion; whenever two or more players become connected they exchange their words and update the list of duplicate words and their scores accordingly.

an open issue for ad hoc networks. Detection of connectivity changes is another challenge. Implementation is relatively easy in the case of mobile agents, but is constrained by the capability of the APIs with which the hardware mobile hosts are equipped. As for logical mobility, Lime should ideally provide a communication API that can be exploited by any Java-based mobile code system—very poor primitives exist today. Minimizing the invasiveness of Lime on the host mobile code system is another complex design task. All these issues entail a careful balancing act between semantic simplicity and implementability. For instance, Mobile UNITY has been used to define a number of interesting and potentially useful constructs but it is unclear at this time the extent to which they have reasonable implementations in the presence of arbitrary disconnections. There are also many interesting and still unexplored constructs, e.g., transient sharing of complex data structures. What are the weakest assumptions one needs to make while still able to ensure implementability and usefulness? This is a question that we will continue to ask ourselves throughout this investigation.

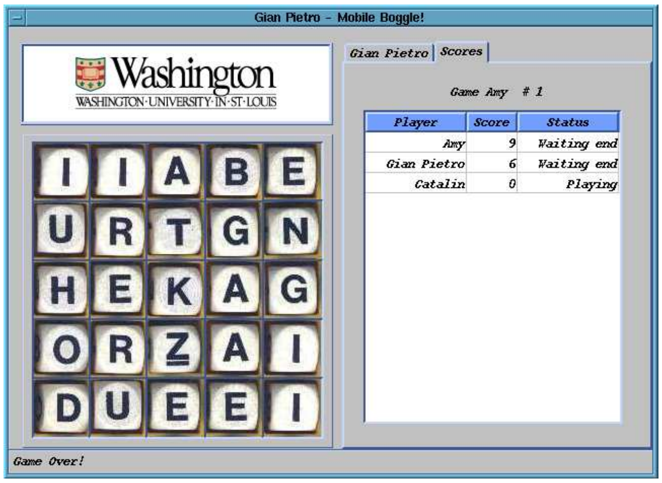

Figure 7: Another snapshot of Mobile Boggle. Game termination is determined by connectivity. If a player's timer has expired, this information is passed to the other players whenever they become connected. When a player knows that all the other players have terminated, the final scores are computed. In the figure, Gian Pietro knows Amy has terminated but Catalin has not.

The soundness of our approach is going to be verified during application development. Applicability to real-world mobile devices will be a driving criteria in selecting the domains we will experiment with. We plan to experiment with a variety of hardware including laptops, hand-held devices, and possibly Java-enhanced customer appliances, in order to evaluate their different requirements with respect to platform support, computational resources, and usability. Applications that can benefit from Lime are countless. Transient sharing could enable hand-held devices to re-synchronize their data with central repositories without requiring insertion in docking stations: being in range with the source of information would be sufficient for triggering a reactive exchange of information. A team of robots at the scene of natural disaster can build cooperatively and dynamically a global map of the area by transiently sharing their local information each time they come in contact. Finally, an intelligent transport system can feed drivers on a highway location-dependent information with the transfer being activated by the appearance of a car in the communication range, or by an explicit request made by mobile agents on behalf of the driver. Our initial set of target applications under development focuses on cooperative games that exhibit predictable disconnection and applications that involve mobile agents. Figures 6 and 7 show snapshots of a LIME implementation of a popular word game[1], called Boggle^{TM}.

A prototype of LIME, built upon IBM's T-Spaces [5] is under development at our university. The initial version was developed under the constraining assumptions of full connectivity and anticipated disconnection—indeed, meaningful for small communities of people using collaborative applications in a short communication range, e.g., managers synchronizing their agendas in an airport meeting room. The next objective is to build a Java API fully implementing the LIME abstractions. This will give us the opportunity to evaluate the implementability of our constructs and possibly modify the design of LIME accordingly. At the same time, this will provide a software package available for experimental development of sample applications involving both physical and logical mobility.

8 CONCLUSIONS

In this paper we presented the design of LIME, a system that adapts the Linda model of communication to mobility by introducing the notions of transiently shared tuple spaces, tuple location, and reactive statement. The hypothesis behind our research is that the minimalist set of operations and concepts provided by LIME, in particular the transient sharing of tuple spaces, enable rapid and dependable development of applications which involve mobility.

Our ultimate goal is to verify the validity of this hypothesis. The approach we plan to pursue involves a blend of formal studies, algorithm design, and application development. We seek to provide application developers with tools that facilitate rapid development of such applications through careful selection of a small set of communication primitives and constructs which have semantic definitions tailored for mobility, support both decoupled and cooperative styles of computing, are precisely defined so as to enable formal analysis, and ensure a high degree of dependability with minimal programming effort. We see LIME as the key vehicle for achieving these ambitious goals with Mobile UNITY providing the formal underpinnings in terms of semantic definitions, verification methods, and novel coordination constructs.

[1] Boggle^{TM} is a trademark of Parker Brothers.

ACKNOWLEDGEMENTS

This paper is based upon work supported in part by the National Science Foundation (NSF) under grants No. CCR-9217751 and CCR-9624815. Gian Pietro Picco was partially supported by Centro Studi e Laboratori Telecomunicazioni (CSELT) S.p.A., Italy. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or CSELT. The authors wish also to thank Gustavo A. Rosini for his implementation of the Boggle demo and his suggestions and comments about LIME.

REFERENCES

[1] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[2] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, May 1998.

[3] D. Garlan and D. Le Métayer, editors. *Proc. of the 2^{nd} Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, volume 1282 of *LNCS*. Springer, Sept. 1997.

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

[5] IBM. T Spaces. `www.almaden.ibm.com/cs/TSpaces`.

[6] D.B. Johnson. Routing in Ad Hoc Networks of Mobile Hosts. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 158–163, 1994.

[7] T.M. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, Mar. 1994.

[8] P.J. McCann and G.-C. Roman. Compositional Programming abstractions for Mobile computing. *IEEE Trans. on Software Engineering*, Feb. 1998.

[9] Sun Microsystems. *JavaSpace Specification*, March 1998. `http://java.sun.com/products/jini/specs`.

[10] C. Perkins. IP mobility support. RFC 2002, IETF Network Working Group, 1996.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline.* Prentice Hall, 1996.

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