Software Architecture for Mobile Computing

Amy L. Murphy[1], Gian Pietro Picco[2], and Gruia-Catalin Roman[3]

[1] University of Rochester, Rochester NY 14607, USA,
murphy@cs.rochester.edu,
http://www.cs.rochester.edu/u/murphy
[2] Politecnico di Milano, Milan, Italy,
picco@elet.polimi.it
http://www.elet.polimi.it/~picco
[3] Washington University, St. Louis MO 63130, USA,
roman@cse.wustl.edu
http://www.cse.wustl.edu/~roman

Abstract. One form of software architecture is a framework for systems that serve the needs of a specific domain. These frameworks must contain sufficient detail to not lose the interesting aspects of the environment, yet they must not expose so many details as to be overwhelming and force the developer to lose the big picture. As the environments we develop for become more complex, it becomes more necessary to compose these frameworks in order to manage the complexity. Mobility is precisely one such environment that is emerging as computing components shrink in size and become more portable. As these components change location in space, their connectivity to other components changes and thus their access to data changes. Some programs needs to be able to respond to this change in connectivity. Others are able to abstract it away, simply perceiving changes in connectivity as changes in data availability. In this paper, we overview a solution to managing the complexity of applications for the the mobile environment in the context of a middleware. First, we present a meta-model, or a framework for generating middleware for mobile environments. Second, we show how this meta-model has been instantiated in the LIME middleware and how it has been used to develop several mobile applications.

1 Introduction

Mobility entails the study of systems in which components change location, in a voluntary or involuntary manner, and move across a space that may be defined to be either logical or physical. By definition, systems of mobile components are distributed systems, and while distributed computing has been carefully studied for decades, mobility poses new challenges that have not previously been addressed.

The development of compact computing devices such as notebook computers and personal digital assistants allow people to carry computational power with them as they change their physical location in space. The number of such components is steadily increasing. One goal, referred to as ubiquitous computing [21], is

for these devices to become seamlessly integrated into the environment until we are no longer explicitly aware of their presence, much the way that the electric motor exists in the world today. Part of enabling this vision is coordinating the actions of these devices, most likely through wireless mediums such as radio or infrared.

Logical mobility, or the movement of code and state through a fixed infrastructure of servers, is emerging as a powerful design abstraction for distributed systems. The pervasiveness of the Java programming language and its portability have led to a wealth of mobile agent systems. Demonstration purpose applications built on top of these systems range from logical agents managing physical objects in a kitchen [11] to agents managing the placement of a video conferencing server to minimize bandwidth consumption [1].

Developing applications in the mobile environment is a difficult task. Many existing applications restrict themselves to addressing a specific aspect of mobility in a highly specialized environment, such as disconnected operation in the Coda filesystem [8] or using agents to perform remote queries on a database as in the Oracle Agent System [13]. Development of these systems requires highly specialized knowledge of low level networking as well as details of the application domain.

Our goal is to enable the development of diverse classes of applications by providing flexible abstractions that can be applied in a variety of settings. Our success in this area comes from an integrated research approach that involves analyzing the needs of mobile applications, formulating models to describe the key concepts of our approaches, specifying formally these models, implementing the abstractions, and returning to the development of applications to evaluate our results.

Our work focuses on mobile ad hoc networks where no infrastructure exists to support communication among physically mobile hosts. Instead, hosts communicate directly with one another and the distance between hosts determines connectivity. A system is typically composed of multiple groups of hosts with connectivity available within the group but no communication from one group to another. Changes in connectivity and corresponding changes in available resources make this a challenging environment for application design.

Our strategy for development in this arena is the design for new high-level coordination abstractions, generically referred to as *global virtual data structures*. The abstraction presented to the application programmer is simply a local data structure whose content changes according to connectivity. Conceptually each component stores a piece of a global data structure, when components are within communication range these pieces are transiently shared and accessible to other components. Interaction with the data structure occurs exclusively by executing operations on the local data structure, however, transient sharing enables transparent interaction with other mobile components.

One of the features of this approach is its ability to facilitate the development of applications that never explicitly access remote data. We term this *context-transparent interaction*, where the data is part of the current context in

which a mobile component finds itself. The distribution and changes to the data structure are hidden from the application programmer by the abstraction itself. Alternately, *context-aware interaction* can easily be provided as an extension to the basic model by explicitly introducing the notion of location.

We have successfully applied this strategy in the development of LIME, Linda In a Mobile Environment, which provides the simple mobile coordination abstraction through transiently shared Linda tuple spaces, enabling application programmers to clearly separate the concerns of computation from the communication among hosts. The implementation of LIME in the form of middleware presents the same interface and semantics as the model, simplifying the implementation process. Mobile application developers utilizing the LIME concepts need not concern themselves with any of the low level details of communication or changing connections, as all of these are handled within the implementation of the middleware.

Work with the LIME system has shown it to be a clean conceptual tool for introducing programmers to the concepts of mobility. Several applications have been built on top of the middleware, demonstrating its usefulness in a variety of mobility scenarios.

In this paper, Section 2 provides an introduction to the concept of global virtual data structures, Section 3 describes the instantiation of this concept in the LIME coordination model, Section 4 steps through an application that sits on top of LIME, and Section 5 concludes with future directions for this work.

2 Global Virtual Data Structures

Physical mobility through space can be categorized into base station mobility and ad hoc mobility. Base station mobility is similar to the cellular telephone system, where mobile components (i.e., mobile telephones) communicate with one another and with the fixed network by always communicating first with a base station (i.e., cellular tower). Ad hoc mobility distinguishes itself from base station mobility by completely removing the fixed infrastructure, leaving only direct communication among hosts. In a mobile ad hoc network, the distance between components determines connectivity. As components move, the system is continuously reshaped into multiple partitions, with connectivity available within each partition but not across partitions.

Freeing mobile users from a fixed infrastructure makes the ad hoc network model ideal for many scenarios such as systems of small components with limited resources to spend on communication, situations in which the infrastructure has been destroyed such as following a natural disaster, and for settings in which establishing an infrastructure is impossible as in a battlefield environment or economically impractical as in a short duration meeting or conference.

The application needs in these scenarios can be classified broadly by how they interact with their changing environment, or *context*. The context of a mobile unit consists of two primary components: system configuration and data. System configuration context describes the knowledge about which mobile units

are connected and possibly also about topology information concerning physical location in space or logical connectivity. This knowledge is limited to the current partition of the network in which the mobile unit finds itself. We refer to this as the current transient group. Because communication cannot extend beyond the group, knowledge of configuration beyond the boundaries of the group is not possible. Data context refers to the more passive data elements and resources that are carried by the mobile components.

This view of context fosters two distinct programming styles: *context aware programming* and *context transparent programming*. Context aware applications are those that access both the system configuration context and the data context explicitly. For example, a context aware application may store a piece of new data on a specific mobile host, or retrieve a piece of data from a named mobile host. All operations must be carried out within the current connectivity context, but this style is distinguished by the needs of the application to be aware of the current context. In contrast, context transparent applications can be developed without explicit knowledge of the current context. Data access is performed on the data in the current context without regard to where it is located. Such applications do not need to be aware of the details of the configuration changes, but simply aware that they are occurring and that these changes affect the available resources. Many applications require a combination of both context aware and context transparent programming.

Our goal is to enable the rapid and dependable development of both styles of application programs for the mobile ad hoc environment. Fundamentally our approach is to design abstractions tailored to the ad hoc environment that hide many of the unnecessary details, but give the programmer sufficient power to tailor the abstraction to their specific needs. This involves providing both context aware and context transparent operations within the same abstraction. At the same time, implementations of these abstractions must be responsive to the technical challenges of the environment.

Our approach to abstractions to simplify the programming task comes from a study of coordination models for distributed computing that separate the computation, or the task-specific programming, from the communication, or the interaction among processes. Distributed coordination models also consider the need to take local decisions while still conceptualizing the effect of these actions on the global scale. Thus, our driving design strategy can be summarized by the desire to coordinate mobile ad hoc applications by thinking globally but acting locally.

2.1 The GVDS Model

One common coordination mechanism in distributed systems is shared memory, or more structured shared data structures. Through this, the complexity of large systems is managed by accessing a single, global data structure. An implementation may be distributed, but the user is not aware of this. The concept of shared memory is appealing in the mobile environment, which is itself a distributed

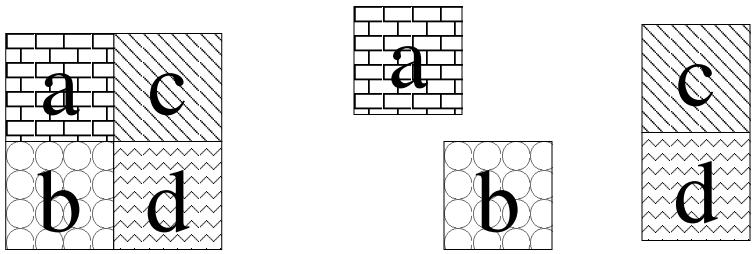

Fig. 1. Transforming a matrix into a global virtual data structure by distributing it among mobile units.

system, however disconnections and the resulting inaccessibility of data make a direct application of shared memory to mobile systems impossible.

By applying our design strategy to shared memory data structures the global data structure emerges as the concept we wish to conceive of globally, but connectivity does not allow this. The first step toward a mobility-viable global data structure is to make explicit the distribution of data across the mobile components, or mobile hosts. For example, Figure 1(a) shows how a large matrix can be evenly divided among four hosts. While all mobile components are within communication range, the entire structure remains accessible to all processes.

When mobile components move and connectivity changes, the available portions of the data change to reflect only reachable data. In Figure 1(b) mobile components a and b are each isolated from all other components, restricting access only to their local partitions of the matrix. However, components c and d remain connected to one another and have access to the combination of c's and d's data.

The global matrix data structure of Figure 1(a) can be visualized at any time from outside the system by ignoring connectivity constraints and combining the data from all mobile components. This is, however, only a *virtual* data structure because it cannot be created in reality. Despite this it remains a powerful concept to the programmer to view how local changes affect the entire system.

We have discussed how connectivity limits the availability of data, but we must also consider how the operations that access the data structure change in response to this accessibility constraint. Some operations must clearly be restricted if full connectivity is not available. In the matrix example, computations such as matrix inversion require the entire matrix and must be restricted. Many operations, however, require no changes and can simply be evaluated over the current projection of the global virtual data structure. These operations play an important role in implementing context-transparent applications as they do not require the programmer to be aware of the details of the environment, but are simply aware that it is changing. Finally some operations can be extended to explicitly address the distribution of data over the hosts. Consider an alternate

division of the matrix example that distributes data based on some aspect of the data other than its location in the matrix. In this case, it may be meaningful to query the part of the distributed data located at a specific agent. These operations are likely to play a role in context-aware applications.

For any global virtual data structure to be successful, its development cycle must include not only the model definition, but also formal specification and implementation. The informal model presents the underlying data structure, how it changes with respect to connectivity, what the primitives are, and how they are affected and extended. Most importantly, the informal model also describes the abstraction provided to the programmer and a way of thinking to effectively develop applications on top of the model. Next, formal semantics force clear definitions of all model concepts and how they are affected by mobility before beginning an implementation. The formal specification also enables user applications to be formally specified and reasoned about, lending dependability to the resulting system. Finally, the data structures must be implemented and applications built. One mechanism to deliver the data structures is via a middleware that sits between the application and the operating system, providing the abstractions defined by the model and formal specification.

The key to development from these three key perspectives is to allow each step to inform the others in an iterative fashion. By considering the needs of the applications, the primitives of the model can be defined and extended to meet the demands of the application programmer. A formal specification can reveal key parts in the model where restrictions must be made to keep the operations computable in the presence of disconnections. The formal specification also informs the implementation, showing where the complexity is involved in the interactions of concurrent programs. A proper implementation must adhere to the formal specification. The process of implementing may reveal atomicity assumptions of the model that are either impossible or impractical to implement. This can lead to an expansion of the model to include more elements of the environment, or to a weakening of the model constructs to make them more practical. Complementary changes must also be made to the formal specification.

2.2 Instantiating a GVDS

Many standard distributed data structures have the potential to be converted into global virtual data structures. For each structure, the fundamental issues to address as part of the evaluation and development processes are similar: Does the data structure match the basic needs of the underlying application? Is there a natural and useful partitioning of the data structure across units in a mobile ad hoc network? How is the data structure perceived by the individual units as changes in connectivity occur?

A tree, as in Figure 2, could be partitioned among units with the nodes where a cut occurs being replicated. A global naming convention would allow communicating units to determine the relation between the tree fragments they carry and make content and structural changes (e.g., swapping subtrees) as long as no disconnected units are affected. In principle, certain operations (e.g., adding a

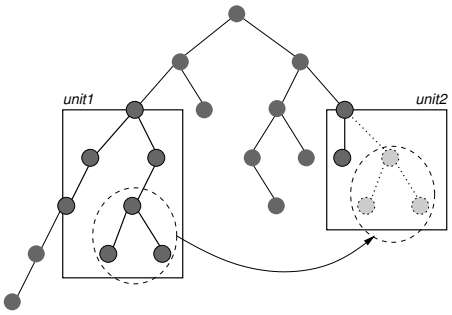

Fig. 2. A hierarchical data structure where units in range agree to transfer a subtree that is under their jurisdiction even though parts of the global structure remain hidden. Moving a subtree distributes data to a different location to satisfy changing access patterns.

leaf node) could be issued at any time with their evaluation being delayed until such time that the affected units are within range. Attempts to access nodes on disconnected units may result in blocking the respective agent. The generalization to a directed graph is straightforward and can overcome the problems caused by the possible loss of one of the units.

Other data structures may be devised to meet the needs of highly specialized applications. For instance, resource-limited units searching a physical space may appear logically as ants crawling on a fixed network of passageways (Figure 2.2). Each unit's knowledge of the surrounding geography is enhanced by the knowledge of all the other units within range. As the density of units decreases, each unit must maintain more and more information. Finally, at a point when the unit's memory is full, information needs to be dropped, e.g., only the main passageways are kept. In an application involving the construction of distributed predictive models of the changes taking place in a physical environment it is conceivable to have the units tied together by a complex structure that combines information about space and time. Each unit may be exploring and collecting data in the present while simulating the future in order to build a predictive model. As units meet they may exchange information about the present but also about various points in the future since some units may be further ahead than others in their simulation.

In the field of parallel programming, tuple space communication à la Linda provides a good example of how coordination can simplify the programming task. Tuple space coordination facilitates temporal and spatial decoupling among parallel programs. By limiting the power of the tuple space access primitives, efficient implementation is achieved as well. The programmer is presented with the appearance of a persistent global data structure that can be readily understood and operated on: a set of tuples accessed by content. Applying the concept of global virtual data structures to Linda yields a model that distributes the global

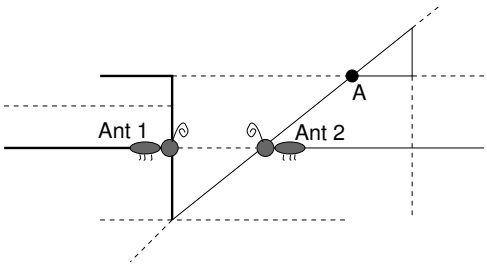

Fig. 3. Ant 1 learns from Ant 2 about landmark A when, by virtue of being in range, the locally built maps are merged. Solid lines denote paths explored by Ants 1 and 2, and dashed lines denote unexplored regions. After sharing, each ant has the same knowledge of the global structure.

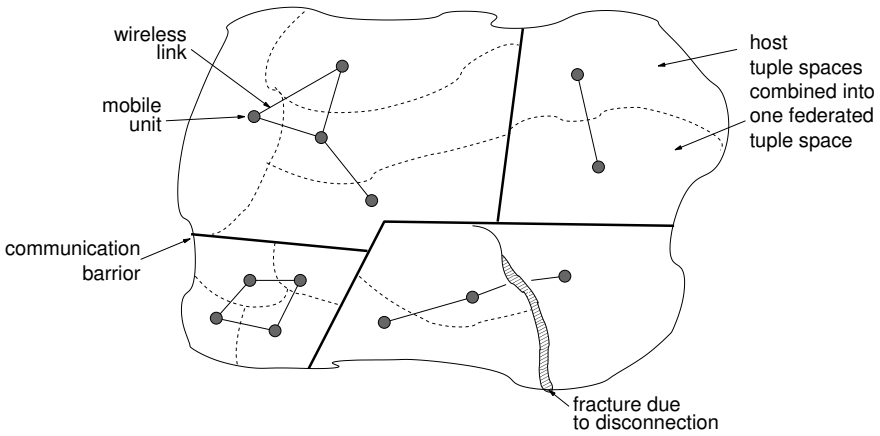

Fig. 4. Creating the illusion of a globally shared tuple space.

tuple space among the mobile units and limits access to the confines of each mobile ad hoc network. For a programmer, mobility is perceived simply as an independently evolving host tuple space, i.e., a continuously changing context. When the mobile components are co-located, the tuple spaces are transiently shared and all tuple space accesses, including pattern matching for reading and removing data, are done on the now shared data space (see Figure 4). Additional primitives with extensions for location are straightforward to access specific tuple spaces, however the presence of the specified tuple space is dependent on connectivity. This data structure has been explored in detail and has resulted in the LIME model, Linda in a Mobile Environment. The details of LIME are presented in the next section.

3 LIME, a GVDS

The LIME model [15, 12] is the first full instantiation of the GVDS concept, and as such, it provides a proof of concept of the idea itself. LIME borrows and adapts the communication model made popular by Linda [4] to provide a coordination abstraction for the mobile environment. After presenting a concise Linda primer, the remainder of this section discusses how the core concepts of Linda are reshaped in the LIME model and embodied in the programming interface of the corresponding middleware implementation.

3.1 Linda in a Nutshell

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

Tuples are added to a tuple space by performing an **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[4]. Moreover, some variants of Linda (e.g., [19]) provide also *bulk operations*, which can be used to retrieve all matching tuples in one step. In LIME we provide a similar functionality through the **ing** and **rdg** operations, whose execution is asynchronous like in the case of probes[5].

3.2 The LIME Model

Linda characteristics resonate with the mobile setting. In particular, communication in Linda is decoupled in *time* and *space*, i.e., senders and receivers do not

[4] Additionally, Linda implementations often include 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.

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

need to be available at the same time, and mutual knowledge of their identity or location is not necessary for data exchange. This form of decoupling is of paramount importance in a mobile setting, where the parties involved in communication change dynamically due to their migration or connectivity patterns. Moreover, the notion of tuple space provides a straightforward and intuitive abstraction for representing the computational context perceived by the communicating processes. On the other hand, decoupling is achieved thanks to the properties of the Linda tuple space, namely its global accessibility to all the processes, and its persistence—properties that are clearly hard if not impossible to maintain in a mobile environment. Finally, these properties make Linda tuple spaces amenable to providing the basis for the GVDS meta-model.

The Core Idea: Transparent Context Maintenance In Linda, the data accessible through the tuple space represents the data *context* available during process interaction. In the model underlying LIME, the shift from a fixed context to a dynamically changing one is accomplished by breaking up the Linda tuple space into many tuple spaces, each permanently associated to a mobile unit, and by introducing rules for transient sharing of these individual tuple spaces based on connectivity.

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

When multiple mobile units are able to communicate, either directly or transitively, we say these units form a LIME *group*. We can restrict the notion of group membership beyond simple communication, but for the purposes of this paper, we consider only connectivity. Conceptually, the contents of the ITSs of all group members are merged, or transiently shared, to form a single, large context that is accessed by each unit through its own ITS. The sharing itself is transparent to each mobile unit, however as the members of the group change, the content of the tuple space each member perceives through operations on the ITS changes in a transparent way.

The joining of a group by a mobile unit, and the subsequent merging of its local context with the group context is referred to as *engagement*, and is performed as a single, atomic operation. A mobile unit leaving a group triggers *disengagement*, that is, the atomic removal of the tuples representing its local context from the remaining group context. In general, whole groups can merge, and a group can split into several groups due to changes in connectivity.

In LIME, agents may have multiple ITSs distinguished by a name since this is recognized [2] 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

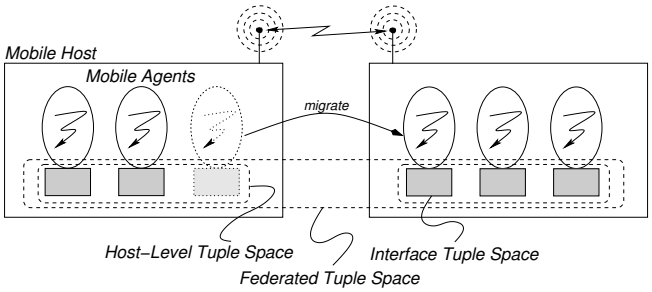

Fig. 5. Transiently shared tuple spaces encompass physical and logical mobility.

group. Thus, for instance, when an agent a owning a single tuple space named X joins a group constituted by an agent b that owns two tuple spaces named X and Y, only X becomes shared between the two agents. Tuple space Y remains accessible only to b, and potentially to other agents owning Y that may join the group later on.

Transient sharing of the ITS constitutes a very powerful abstraction, as it provides a mobile unit with the illusion of a local tuple space that contains all the tuples coming from all the units belonging to the group, without any need to know the members explicitly. The notion of transiently shared tuple space is a natural adaptation of the Linda tuple space to a mobile environment. When physical mobility is involved, and especially in the radical setting defined by mobile ad hoc networking, there is no stable place to store a persistent tuple space. Connections among machines come and go and the tuple space must be partitioned in some way. Analogously, in the scenario of logical mobility, maintaining locality of tuples with respect to the agent they belong to may be complicated. LIME enforces an a priori partitioning of the tuple space in subspaces that get transiently shared according to precise rules, providing a tuple space abstraction that depends on connectivity.

Encompassing Physical and Logical Mobility In LIME, mobile hosts are connected when a communication link is available. Availability may depend on a variety of factors, including quality of service, security considerations, or connection cost; all of which can be represented in LIME, although in this paper we limit ourselves to availability determined by the presence of a functioning link. Mobile agents are connected when they are co-located on the same host, or they reside on hosts that are connected. Changes in connectivity among hosts depend only on changes in the physical communication links. Connectivity among mobile agents may depend also on arrival and departure of agents, with creation and termination of mobile agents being regarded as a special case of connection and disconnection, respectively. Figure 5 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 with them.

The transiently shared ITSs belonging to multiple agents co-located on a host define a *host-level tuple space*. The concept of transient sharing can also be applied to the host-level tuple spaces of connected hosts, forming a *federated tuple space*. When a federated tuple space is established, a query on the ITS of an agent returns a tuple that may belong to the tuple space carried by that agent, to a tuple space belonging to a co-located agent, or to a tuple space associated with an agent residing on some remote, connected host.

In this model, physical and logical mobility are separated in two different tiers of abstraction. Nevertheless, many applications do not need both forms of mobility, and straightforward adaptations of the model are possible. For instance, applications that do not exploit mobile agents but run on a mobile host can employ one or more stationary agents, i.e., programs that do not contain migration operations. In this case, the design of the application can be modeled in terms of mobile hosts whose ITS is a fixed host-level tuple space. Applications that do not exploit physical mobility—and do not need a federated tuple space spanning different hosts—can exploit only the host-level tuple space as a local communication mechanism among co-located agents.

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

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 unit is provided in LIME by extending the Linda operations with tuple location parameters that operate on user-defined projections of the transiently shared tuple space. Further, all tuples are implicitly augmented with two fields, representing the tuple's *current* and *destination*

location. The current location identifies the single agent responsible for holding the tuple when all agents are disconnected, and the destination location indicates the agent with whom the tuple should eventually reside.

The **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 calling the operation, say ω. At this point the tuple t has a current location ω, and a destination location λ. If the agent λ is currently connected, the tuple t is moved to the destination location in the same atomic step. On the other hand, if λ is currently disconnected the tuple remains at the current location, the tuple space of ω. This *"misplaced"* tuple, if not withdrawn[6], will remain misplaced unless λ becomes connected. In the latter case, the tuple will migrate to the tuple space associated with λ as part of the engagement. By using **out**$[\lambda]$, the caller can specify that the tuple is supposed to be placed within the ITS of agent λ. This way, the default policy of keeping the tuple in the caller's context until it is withdrawn can be overridden, and more elaborate schemes for transient communication can be developed.

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

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

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

[6] 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, it might be the case that some other agent may withdraw t from the tuple space before λ, even after t reached λ's ITS.

Reacting to Changes in Context In the fluid scenario we target, the set of available data, hosts, and agents change rapidly according to the reconfiguration induced by mobility. Reacting to changes constitutes a significant fraction of an application's activities. At first glance, the Linda model would seem sufficient to provide some degree of reactivity by representing relevant events as tuples, and by using the **in** operation to execute the corresponding reaction as soon as the event tuple appears in the tuple space. Nevertheless, in practice this solution has a number of drawbacks. For instance, programming becomes cumbersome, since the burden of implementing a reactive behavior is placed on the programmer rather than the system. Moreover, enabling an asynchronous reaction would require the execution of **in** in a separate thread of control, hence degrading performance. Therefore, LIME explicitly extends the basic Linda tuple space with the notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s that specifies the actions to be executed when a tuple matching the pattern p is found in the tuple space. The semantics of reactions are based on the Mobile UNITY reactive statements [10]. Informally, a reaction can *fire* if a tuple matching pattern p exists in the tuple space. After every regular tuple space operation, a reaction is selected non-deterministically and, if it is enabled, the statements in s are executed in a single, atomic step. This selection and execution continues until no reactions are enabled, at which point normal processing resumes. Blocking operations are not allowed in s, as they may prevent the execution of s from terminating.

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

Reactions provide the programmer with very powerful constructs. They enable the specification of the appropriate actions that need to take place in response to a *state* change and allow their execution in a single atomic step. In particular, it is worth noting how this model is much more powerful than many event-based ones [18], including those exploited by tuple space middleware such as TSpaces [6] and JavaSpaces [7], that are typically stateless and provide no guarantee about the atomicity of event reactions.

Nevertheless, this expressive power comes at a price. In particular, when multiple hosts are present, the content of the federated tuple space depends on the content of the tuple spaces belonging to physically distributed, remote agents. Thus, maintaining the requirements of atomicity and serialization imposed by reactive statements requires a distributed transaction encompassing several hosts for every tuple space operation on any ITS—very often, an impractical solution.

For specific applications and scenarios, e.g., those involving a very limited number of nodes, these kind of reactions, referred to as *strong reactions*, would still be reasonable and therefore they remain part of the model. For practical performance reasons, however, our implementation currently limits the use of strong reactions by restricting the current location field to be a host or agent, and by enabling a reaction to fire only when the matching tuple appears on the same host as the agent that registered the reaction. As a consequence, a mobile agent can register a reaction for a host different from the one where it is residing, but such a reaction remains disabled until the agent migrates to the specified host. These constraints effectively force the *detection* of a tuple matching p and the corresponding *execution* of the code fragment s to take place (atomically) on a single host, and hence does not require a distributed transaction.

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

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

Without awareness of the system configuration, only a partial context awareness can be accomplished, where applications are aware of changes in the portion of context concerned with application data. Although this perspective is often enough for many mobile applications, in many others the portion of context more closely related to the system configuration plays a key role. For instance, a typical problem is to react to departure of a mobile unit, or to determine the set of units currently belonging to a LIME group. Interestingly, LIME provides this form of awareness of the system configuration by using the same abstractions discussed thus far: through a transiently shared tuple space conventionally named LimeSystem to which all agents are permanently bound. The tuples in this tuple space contain information about the mobile units present in the group and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, which host they reside on. Insertion and withdrawal of tuples in LimeSystem is a prerogative of the run-time support. Nevertheless, applications can read tuples and register reactions to respond to changes in the configuration of the system.

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

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.

3.3 Programming with LIME

We complete the presentation of the LIME model by concisely illustrating the application programming interface provided in the current implementation[7] of LIME.

The class `LimeTupleSpace`, whose public interface is shown[8] in Figure 6, embodies the concept of a transiently shared tuple space. 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.

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

[7] The LIME Web site [20] contains extensive documentation and programming examples.

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

`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 6). Engagement or disengagement of an entire host, instead, can be triggered explicitly by the programmer by using the methods `engage` and `disengage`, provided by the `LimeServer` class, not shown here. Otherwise, they are implicitly called by the run-time support according to connectivity. The `LimeServer` class is essentially an interface towards the run-time support, and exports additional system-related features, e.g., loading of an agent into a local or remote run-time support, setting of properties, and so on. In particular, it also allows the programmer to define whether transient sharing is constrained to a host-level tuple space, or whether it spans the whole federated tuple space.

`LimeTupleSpace` contains the Linda operations needed to access the tuple space, as well as the operation variants annotated with location parameters. The only requirement for tuple objects is to implement the interface `ITuple`, which is defined in a separate package providing access to a lightweight tuple space implementation. As for location parameters, LIME provides two classes, `AgentLocation` and `HostLocation`, which extend the common superclass `Location`, enabling the definition of globally unique location identifiers for hosts and agents. Objects of these classes are used to specify different scopes for LIME operations, as described earlier. For instance, a probe `inp(cur,dest,t)` may be restricted to the tuple space of a single agent if `cur` is of type `AgentLocation`, or it may refer the whole host-level tuple space, if `cur` is of type `HostLocation`. The constant `Location.UNSPECIFIED` is used to allow any location parameter to match. Thus, for instance, `in(cur,Location.UNSPECIFIED,t)` returns a tuple contained in the tuple space of `cur`, regardless of its final destination, including also misplaced tuples. Note how typing rules allow the proper constraint of the current and destination location according to the rules of the LIME model. For instance, the `destination` parameter is always an `AgentLocation` object, as agents are the only carriers of "concrete" tuple spaces in LIME. In the current implementation of LIME, probes are always restricted to a local subset of the federated tuple space, as defined by the location parameters. An unconstrained definition, as the one provided for `in` and `rd`, would involve a distributed transaction in order to preserve the semantics of the probe across the federated tuple space.

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

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

The remainder of the interface of `LimeTupleSpace` is devoted to managing reactions; other relevant classes for this task are shown in Figure 7. Reactions can either be of type `LocalizedReaction`, where the current and destination location restrict the scope of the operation, or `UbiquitousReaction`, that specifies the whole federated tuple space as a target for matching. The type of a reaction is used to enforce the proper constraints on the registration through type checking. These two classes share the abstract class `Reaction` as a common ancestor, which defines a number of accessors for the properties established for the reaction at creation time. Creation of a reaction is performed by specifying the template that needs to be matched in the tuple space, a `ReactionListener` object that specifies the actions taken when the reaction fires, and a mode. The `ReactionListener` interface requires the implementation of a single method `reactsTo` that is invoked by the run-time support when the reaction actually fires. This method has access to the information about the reaction carried by the `ReactionEvent` object passed as a parameter to the method. The reaction mode can be either of the constants `ONCE` or `ONCEPERTUPLE`, defined in `Reaction`. Reactions are added to the ITS by calling either `addStrongReaction` or `addWeakReaction`, depending on the desired semantics. As we discussed earlier, in the current implementation strong reactions are confined to a single host, and hence only a `LocalizedReaction` can be passed to the first method. Registration of a reaction returns an object `RegisteredReaction`, that can be used to

deregister a reaction with the method `removeReaction`, and provides additional information about the registration process. The decoupling between the reaction used for the registration and the `RegisteredReaction` object returned allows for registration of the same reaction on different ITSs and for the same reaction to be registered with strong and, subsequently, with weak semantics.

4 Application Development

LIME has been used in the development of a variety of mobile applications. In this section, we focus on applications dealing with physical mobility of hosts and first present a high level description of several different applications built on top of LIME, then we go into detail of another application that shows how physical hosts can perform collaborative tasks in the presence of disconnection.

4.1 Three Brief Examples

The first two applications presented here are not stand-alone applications, but instead add an additional layer of abstraction on top of LIME to support the development of mobile applications. The third is a mobile game that exploits the system configuration information available through LIME to react to changes in connectivity.

Because mobility of hosts defines a working environment in which the accessible components is constantly in flux, applications that must avail themselves of services need a mechanism to discover those services in a dynamic manner. A group from Washington University built a Jini-like service discovery mechanism as an application layer on top of LIME [5]. This project 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 at Politecnico di Milano, the LIME tuple space is used to support code mobility by storing Java class bytecode [14]. 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.

The third application exploits the context aware features of LIME. It 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. The display to the players is dominated by a *radar-like* image with an icon of the player in the middle, and icons indicating the current locations of the other connected players. Up-to-date location information is maintained by

each player periodically inserting a tuple into their local tuple space indicating their current location. All other players register a reaction for these location tuples, and are notified when a player moves. When a player disconnects, their icon is changed to indicate their temporary unavailability. This functionality is attained with a single reaction registered on the LimeSystem tuple space whose listener changes the icon of the disconnected player. RedRover also exploits the ability to create multiple tuple spaces for a single application. Location updates are fed to a common tuple space that is shared by all player, but RedRover uses separate team-only tuple spaces to share private information, such as the location of a flag when playing "capture the flag".

4.2 Extended Example: Accessing Shared Data

RoamingJigsaw, is a multi-player jigsaw assembly game. A group of players cooperate in a disconnected fashion on the solution of the jigsaw puzzle. They can construct assemblies independently (e.g., while disconnected), and share intermediate results or acquire pieces from each other when connected. 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 users show the same set of puzzle pieces at this point.

Players can select pieces in the puzzle tray by clicking on them. The visual effect is that the piece outline is highlighted on all users' displays with the color of the selecting player. Selection has deeper consequences. In fact, although all the puzzle pieces are displayed on the tray, a player can make assemblies using only the pieces that she has selected, and that are currently displayed with her color. 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 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 outlined with her color. Nevertheless, the pieces of all players remain visible. The assemblies made by each player during disconnection become visible to the others when connectivity among the players is restored. At this point, the view provided by the user interfaces is reconciled with the changes made during disconnection, and the selection of a piece belonging to a connected player is again possible. Figure 8 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, this still allows the players to work towards

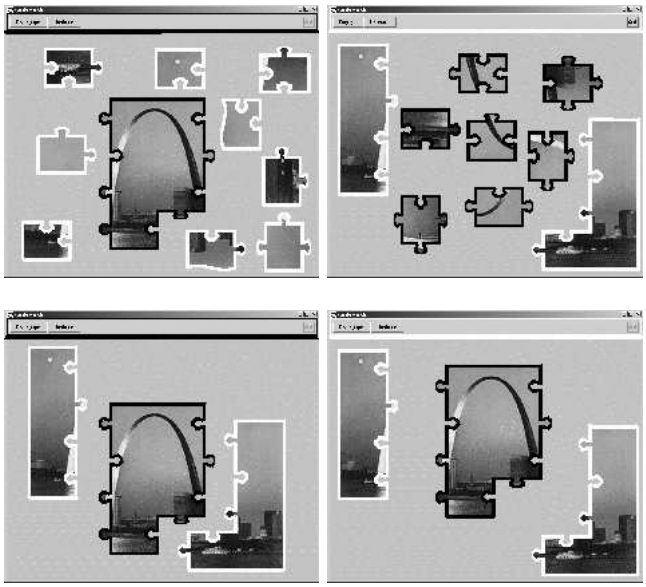

Fig. 8. RoamingJigsaw. The top two images show the puzzle trays of the black and white players while they are disconnected and able to assemble only their selected pieces. The bottom two images show the black and white puzzle trays after the players re-engage and see the assemblies that occurred during disconnection.

achieving 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 could be provided by collaborative work applications involving mobile users, where our mechanism could be used to deal with changes in sections of a document, or with paper submissions and reviews to be evaluated by a program committee.

Design and Implementation. In our design of RoamingJigsaw, we chose to represent pieces and assemblies as tuples, and the shared workspace as a tuple space. When a player selects a piece, the corresponding tuple is withdrawn and subsequently reinserted in the tuple space, with the field indicating the current "owner" automatically changed by Lime. Similarly, when a player builds an assembly out of several pieces, a new tuple is written containing information about the assembled pieces; the tuples associated with the latter are removed from the tuple space.

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 reaction of type `UbiquitousReaction` and mode oncepertuple. Registration of the reaction is specified so that its template looks for any new tuple corresponding to a puzzle piece, while its listener takes care of updating the puzzle tray by using the information found in the tuple, thus correctly maintaining the weakly consistent view of the workspace. Since the reaction type sets its scope to the whole federated tuple space, the application receives updates about new pieces regardless of where and why they have been inserted, and hence notably without any need to be explicitly aware of the arrival and departure of players. Thus, the programming effort can be rightfully spent on handling data changes, rather than monitoring 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 disconnection period, piece selection should actually transfer the corresponding tuple into the local tuple space of the player's application. Moreover, according to what we discussed earlier, a player must be prevented from selecting a piece that is currently not present in the federated tuple space. For this reason, selection is performed by the application agent 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. Otherwise, if no tuple is returned it means that the piece is unavailable for selection, and a message is displayed to the user.

Design Process. The Lime version of RoamingJigsaw was developed as a port of a previous version written on top of the TSpaces middleware [6]. In this version, all puzzle pieces were held at the tuple space server and players issued remote operations. Porting the application to the mobile environment and Lime involved only minor changes to the application, including the introduction of puzzle piece ownership and the conversion of TSpaces clients to Lime agents.

Interestingly, the coordination necessary to handle the inaccessibility of tuples due to disconnection was already addressed in the original application. In the original, when two pieces are assembled, two independent **inp** operations are performed to remove the separate pieces, following by a single **out** to insert the joined piece. If one of the original two pieces is not present (i.e., the **inp** returns `null`), the non-mobile application assumes that some other player is attempting to assemble the same piece simultaneously, and therefore the player backs-off, allowing the other player to continue. If the conflict occurs on the second piece removed, then the first removed piece must be reinserted. The same problem occurs in the mobile version, and similar corrective action is required. Also in the mobile version, a similar issue arises when a player tries to select a piece to become the owner. This operation involves an **inp** that may fail either because another player is trying to select the same piece or because the piece is not accessible due to disconnection. The significance of this is that the programmer of the

mobile version had already encountered complex coordination issues during the development of the server version, and the mobile issues were much the same.

Finally, in converting from TSpaces to LIME, the event mechanisms were changed. TSpaces uses events that fire in response to an operation on the tuple space. Therefore, in order to update a player's puzzle tray, an event was registered on the insertion (i.e., **out**) of a tuple. In LIME, reactions are registered on the state of the tuple space. By replacing the original TSpaces event with a LIME ONCEPERTUPLE reaction, we achieved the same functionality, and simultaneously were able to update the player puzzle trays to reflect changes that occurred during disconnection.

5 Conclusions and Future Directions

Mobility is emerging as an important area for computing research, posing many challenges that must be overcome in a society that is increasingly placing demands on computing technology. Our research into methods for designing middleware for mobile computing, specifically the instantiation of the global virtual data structures concept in LIME has demonstrated the benefit of providing high level abstractions to application developers, easing the software development process and ultimately resulting in reliable applications built on top of a stable platform.

Future work remains to be done in adapting other data structures to the GVDS concept, although some work has already proceeded in this direction. For example, the XMIDDLE [9] system developed at University College of London 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. After disconnection, operations on replicated data are still allowed, and their effect is reconciled when connectivity is restored. Also PEERWARE [3], a project at Politecnico di Milano, exploits a tree data structure, albeit in a rather different way. In PEERWARE, each host is associated with a tree of document containers. When connectivity is available, the trees are shared among hosts, meaning that the document pool available for searching under a given tree node includes the union of the documents at that node on all connected hosts. We are also working on a parallel project to formalize the GVDS concept, identifying the core concepts, making it more accessible to other researchers, and clarifying the process of instantiating the model.

LIME itself is a promising middleware that has taken on a life of its own outside the GVDS model. While the version LIME described here has already been shown to be useful for a variety of applications, and is general enough to provide a foundation for additional mobile ad hoc services, the model itself makes strong guarantees about connectivity that are not always possible in the mobile ad hoc environment. For example, even by incorporating the notion of *safe distance* [17] as part of the engagement and disengagement protocols, it is still possible for a host to disconnect without prior warning. Work is continuing

on LIME to weaken the model to both handle unannounced disconnection and to remove the transactional nature of engagement. We expect this weakening to result in an implementation which is widely applicable, but for which guarantees are difficult to formally describe and even to achieve. We have also begun to explore the issues of security in tuple space based mobile ad hoc environments [16] by allowing applications to protect selected tuple spaces and even individual tuples through the use of passwords. The same passwords are also used to encrypt communication among hosts when exchanging messages related to sharing specific tuples spaces.

Finally, LIME, in addition to demonstrating the practical use of coordination technology in mobile computing, opens a new area of research involving the application of state-based coordination models and middleware to context-aware computing. The complex mobile environment becomes manageable with the abstractions provided by the middleware, the software development process is simplified, and the resulting applications are more reliable.

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

References

1. M. Baldi and G.P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20th Int. Conf. on Software Engineering*, 1998.
2. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus-Linda. In *Workshop on Languages and Models for Coordination, European Conference on Object Oriented Programming*, 1994.
3. G. Cugola and G.P. Picco. PEERWARE: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, Italy, 2001. Available at www.elet.polimi.it/upload/picco.
4. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
5. R. Handorean and G.-C. Roman. Service provision in ad hoc networks. In F. Arbab and C. Talcott, editors, *Proceedings of the 5th International Conference on Coordination Models and Languages*, LNCS 2315, pages 207–219, York, UK, April 2002. Springer.
6. IBM. TSpaces Web page. http://www.almaden.ibm.com/cs/TSpaces.
7. JavaSpaces. The JavaSpaces Specification web page. http://www.sun.com/jini/specs/js-spec.html.
8. J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.
9. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A data-sharing middleware for mobile computing. *Kluwer Personal and Wireless Communications Journal*, 21(1), April 2002.
10. P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2):97–110, 1998.

11. N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed Agents for Networking Things. In *Proc. of the 1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents (ASA/MA '99)*, pages 118–129, Palm Springs, CA, USA, October 1999. IEEE Computer Society.
12. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.
13. Oracle. Oracle 8*i* Lite web page. http://www.oracle.com/, 1999.
14. G.P. Picco and M.L. Buschini. Exploiting transiently shared tuple spaces for location transparent code mobility. In F. Arbab and C. Talcott, editors, *Proc. of the 5th Int. Conf. on Coordination Models and Languages*, LNCS 2315, pages 258–273, York, UK, April 2002. Springer.
15. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.
16. G.-C. Roman and R. Handorean. Secure Sharing of Tuple Spaces in Ad Hoc Settings. Technical Report WUCS-02-31, Dept. of Computer Science and Engineering, Washington Univ. in St. Louis, MO, USA, 2003.
17. G.-C. Roman, Q. Huang, and A. Hazemi. Consistent group membership in ad hoc networks. In *Proceedings of the 23rd Int. Conf. on Software Engineering*, pages 381–388, Toronto, Canada, May 2001.
18. D.S. Rosenblum and A.L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the 6th European Software Engineering Conf. held jointly with the 5th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE97)*, number 1301 in LNCS, Zurich (Switzerland), September 1997. Springer.
19. A. Rowstron. WCL: A coordination language for geographically distributed agents. *World Wide Web Journal*, 1(3):167–179, 1998.
20. Lime Team. LIME Web page. lime.sourceforge.net.
21. M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.