LIME: A Middleware for Physical and Logical Mobility

Amy L. Murphy
Dept. of Computer Science
University of Rochester
P.O. Box 270226
Rochester, NY, 14627, USA
murphy@cs.rochester.edu

Gian Pietro Picco
Dip. di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci, 32
20133 Milano, Italy
picco@elet.polimi.it

Gruia-Catalin Roman
Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130, USA
roman@cs.wustl.edu

Abstract

LIME *is a middleware supporting the development of applications that exhibit physical mobility of hosts, logical mobility of agents, or both.* LIME *adopts a coordination perspective inspired by work on the Linda model. The context for computation, represented in Linda by a globally accessible, persistent tuple space, is represented in* LIME *by transient sharing of the tuple spaces carried by each individual mobile unit. Linda tuple spaces are also extended with a notion of location and with the ability to react to a given state. The hypothesis underlying our work is that the resulting model provides a minimalist set of abstractions that enable rapid and dependable development of mobile applications. In this paper, we illustrate the model underlying* LIME*, present its current design and implementation, and discuss initial lessons learned in developing applications that involve physical mobility.*

1. Introduction

Middleware has emerged as a new development tool which can provide programmers with the bene£ts of a powerful virtual machine specialized and optimized for tasks common in a particular application setting without the major investments associated with the development of application-speci£c languages and systems. Given the complexities associated with software involving mobile hosts and agents, middleware is expected to establish itself as an important new technology. This paper addresses middleware for mobility and presents an example of how a new abstract model supporting both physical and logical mobility can be delivered in the form of middleware.

The starting point for our investigation was the notion that a coordination perspective on mobility holds the key to simplifying the development effort. The idea is to eliminate the programmer's need to be concerned with the mechanics of communication among hosts and agents. The interactions among mobile units of any kind are expressed separately from the application processing and are implemented in a transparent manner by the middleware fabric. The middleware presented in this paper (LIME—Linda in a Mobile Environment) explores this idea by providing programmers with a *global virtual data structure*, a Linda-like tuple space whose content is determined by the connectivity among mobile hosts. Individual programs perceive the effects of mobility as behind-the-scene changes in the content of their own local tuple spaces. The resulting middleware is essentially an embodiment of a model of mobility in which coordination takes place via a global tuple space physically distributed among mobile units and logically partitioned according to connectivity among the units.

When viewed in the broader context of mobility, LIME is indeed a new breed of middleware. LIME is general purpose, model-centric, and inclusive of both physical and logical mobility. It provides novel programming constructs in a manner that is sensitive to the constraints imposed by the realities of mobility.

In the remainder of the paper we provide an overview of LIME (Section 2), we examine the implementation strategy (Section 3), and review our experience with several applications developed using LIME (Section 4). The paper concludes with a brief discussion of lessons learned (Section 5) followed by conclusions (Section 6) and references.

2. LIME: Linda in a Mobile Environment

The LIME model [7] aims at identifying a coordination layer that can be exploited successfully for designing applications that exhibit logical and/or physical mobility. LIME borrows and adapts the communication model made popular by Linda [2].

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 parameters, such as ⟨"foo", 9, 27.5⟩, and contains the actual 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*, and its £elds 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, ?¤oat⟩ 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 de£ned 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 **rd** operation. Both **in** and **rd** are blocking, i.e., the process performing the operation blocks until a matching tuple is found in the tuple space. 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[1].

2.1. The LIME model

Linda characteristics resonate with the mobile setting. In particular, communication in Linda is decoupled in *time* and *space*, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their 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. Another key to mobility is that the global context for operations is de£ned by the transient community of mobile units that are currently present. Since these communities are dynamically changing according to connectivity and migration, the context changes as well.

The Core Idea: Transparent Context Maintainance. In the model underlying LIME, the shift from a £xed 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 the individual tuple spaces based on connectivity.

From the perspective of a mobile unit, the only way to access the global context is through a so-called *interface tuple space* (ITS), which is permanently and exclusively attached to the unit itself. The ITS contains tuples the mobile unit is willing to make available to other units, and that are concretely co-located with the unit itself. This represents the only context accessible to the unit when it is alone. Access to the ITS takes place using the Linda primitives already mentioned, whose semantics is basically unaffected. Nevertheless, this tuple space is also *transiently shared* with the ITSs belonging to the mobile units that are currently part of the community. Hence, the content perceived through the ITS changes dynamically in response to changes in the set of co-located mobile units.

Upon arrival of a new mobile unit, tuples in the ITS of the new unit are merged with those, already shared, belonging to the other mobile units, and the result is made accessible through the ITS of each of the units. This sequence of operations, called *engagement*, is performed as a single atomic operation. Similar considerations hold for the departure of a mobile unit, resulting in the *disengagement* of the corresponding tuple space and the removal of data perceived by the remaining units through their ITSs.

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 community, without any need to know them explicitly.

Encompassing Physical and Logical Mobility. Thus far we have glossed over the nature of the mobile unit at hand, that is, we did not specify whether it is a mobile agent moving in logical space or a mobile host roaming the physical space. This is precisely because the LIME notion of a transiently shared tuple space is applicable to a mobile unit regardless of its nature, as long as a notion of connectivity ruling engagement and disengagement is properly de£ned.

In an ad hoc network, LIME mobile hosts are connected when distance between them allows communication. Mobile agents are connected when they are co-located on the same host, or they reside on hosts that are connected. Creation and termination of mobile agents is a special case of connection and disconnection, respectively. Figure 1 depicts the model adopted by LIME. Mobile agents are the only active components; mobile hosts are mainly roaming containers which provide connectivity and execution support for agents. In other words, mobile agents are the only components that carry a "concrete" tuple space with them.

The transiently shared ITSs belonging to multiple agents co-located on a host de£ne a *host-level tuple space*. The concept of transient sharing can also be applied to the host-level tuple spaces of connected hosts, forming the *federated tuple space*. When a federated tuple space is established, a query on the ITS of an agent returns a tuple that may belong indifferently 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.

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

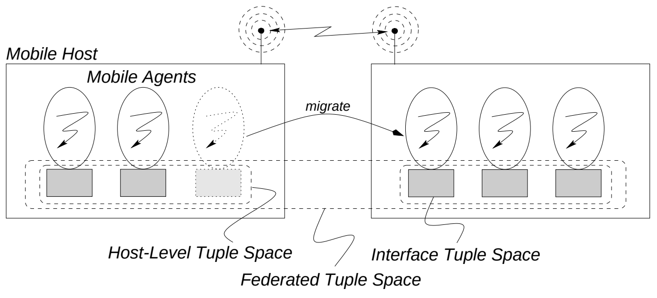

Figure 1. Transiently shared tuple spaces encompass physical and logical mobility.

Controlling Context Awareness. Thus far, LIME appears to foster a style of coordination that reduces the details of distribution and mobility to changes in what is perceived as a local tuple space. This view is powerful as it relieves the designer from speci£cally addressing the changes in con£guration, but some mobile applications need to explicitly address the distributed nature of the data for performance or optimization reasons. Such £ne-grained control over the context perceived by the mobile unit is provided in LIME by extending Linda operations with tuple location parameters that operate on user de£ned projections of the transiently shared tuple space. Tuple location parameters are expressed in terms of agent identi£ers or host identi£ers.

The **out**$[\lambda]$ operation extends **out** with a location parameter representing the identi£er of the agent responsible for holding the tuple. The semantics of **out**$[\lambda](t)$ involve two steps. The £rst 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 atomically moved to the destination location. On the other hand, if λ is not currently connected, the tuple remains at the current location, the tuple space of ω. This "misplaced" tuple, if not withdrawn[2], 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 transaction. By using **out**$[\lambda]$, the caller can specify that the tuple is supposed to be placed within the tuple space 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.

Location parameters also come into play to provide variants of the **in** and **rd** operations to allow access to a slice of the current context. These operations are annotated as **in**$[\omega, \lambda]$ and **rd**$[\omega, \lambda]$, and more details are provided at the end of this section.

Disengagement relies on the notion of tuple location, as well. Upon occurrence of a disconnection, the transiently shared tuple space is partitioned into its constituents, i.e., as if each mobile agent were alone. In this con£guration, the ITS of each mobile agent ω contains only the portion of the transiently shared tuple space it is responsible for, i.e., all the tuples whose current location is ω, including misplaced tuples. Next, the ITS are merged again according to the new system connectivity after disconnection, effectively creating two partitioned federated tuple spaces. It is important to note that the above is simply a conceptual description of the disengagement process. In practice, no tuple transfer is needed to comply with the desired semantics, provided that the atomicity of engagement and of **out**$[\lambda]$ is preserved.

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)con£guration that generated those changes. For instance, if the probe **inp**$[\omega, \lambda](p)$ fails, this simply means that no tuple matching p is available in the projection of the federated tuple space over the location parameters $[\omega, \lambda]$. It is not possible to determine whether the failure is due to the fact that agent ω does not have a matching tuple, or whether agent ω is not currently part of the community.

Without awareness of the system con£guration, only a partial context awareness, that concerned with application data, can be achieved. Although this perspective is often enough for many mobile applications, in some cases the con£guration context plays a key role. For instance, a typical problem is to react to departure of one of the parties involved, or to determine the set of parties currently belonging to the mobile community. LIME exposes this information through a read-only, system-maintained tuple space, conventionally named LimeSystem. Its tuples contain information about the mobile units present in the community, and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, which host they reside on. Standard tuple space operations, including reactions as de£ned next, can be used to access the system con£guration.

Reacting to Changes in Context. Mobility enables a highly dynamic environment, where reaction to changes constitutes a major fraction of the application design. Therefore, LIME extends the basic Linda tuple space with a notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is de£ned by a code fragment s that speci£es the actions to be executed when a tuple matching the pattern p is found in the tuple space. The semantics of reactions is based on Mobile UNITY reactive statements, described in [5]. After each operation on the tuple space, a reaction is selected non-deterministically and

[2]Note how specifying a destination location λ does not imply guaranteed delivery of the tuple t to λ. Linda rules for non-deterministic selection of tuples are still in place; thus, some other agent may withdraw t from the tuple space before λ, even after t reached λ's ITS.

the pattern p is compared against the content of the tuple space. If a matching tuple is found, s is executed, otherwise the reaction is a skip. This selection and execution proceeds until there are no reactions enabled, then normal processing resumes. Thus, reactions are executed as if they belonged to a separate reactive program which is run to £xed point after each non-reactive statement. Blocking operations are not allowed in s, as they may prevent the program from reaching £xed point.

The full form of a reaction is annotated with locations, as in $\mathcal{R}[\omega, \lambda](s, p)$, where the location parameters have the same meaning as discussed for **in** and **rd**. However, these kinds of reactions, called *strong reactions*, are not allowed over the entire federated tuple space; in other words, the current location £eld must always be restricted to a host or agent. The reason for this lies in the constraints introduced by physical mobility. 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 strong reactive statements would require a distributed transaction encompassing several hosts for every tuple space operation on any ITS—very often, an impractical solution.

For these reasons, LIME also provides a notion of *weak reaction*. Weak reactions are used primarily to detect changes in the federated tuple space. In this case, the host where the pattern p is successfully matched to a tuple, and the host where the corresponding action s is executed may be different. Processing of a weak reaction proceeds as in the case of strong reactions, except that the execution of s does not happen atomically with the detection of a tuple matching p: instead, it is guaranteed to take place eventually after such condition, if connectivity is preserved.

2.2. Programming with LIME

We conclude the presentation of the LIME model by brie¤y commenting upon the programming interface that is provided in the current implementation of LIME.

The class `LimeTupleSpace`, shown[3] in Figure 2, embodies the concept of a transiently shared tuple space. Only the thread of the agent which creates the tuple space is allowed to perform operations on the `LimeTupleSpace` object; accesses by other threads fail by returning an exception. This represents the constraint that the ITS must be permanently and exclusively attached to the corresponding mobile agent.

In LIME, agents may have multiple ITSs distinguished by a name, which is the parameter for the `LimeTupleSpace` constructor. The name determines the sharing rule; only tuple spaces with the same name are transiently shared. An

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

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

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

example of keeping information of different tasks and roles separated in multiple tuple spaces is shown in Section 4.

Agents may have also *private* tuple spaces, i.e., not subject to sharing. A private `LimeTupleSpace` can be used as a stepping stone to a shared data space, allowing the agent to populate it with data prior to making it publicly accessible, or it can be useful as a primitive data structure for local data storage. All tuple spaces are initially created private, and sharing must be explicitly enabled by calling the instance method `setShared`. The method accepts a boolean parameter specifying whether the transition is from private to shared (`true`) or vice versa (`false`). Calling this method effectively triggers engagement or disengagement of the corresponding tuple space. Sharing properties for multiple tuple spaces owned by the same agent can also be changed in a single atomic step.

`LimeTupleSpace` contains also the Linda operations needed to access the tuple space, as well as their variants annotated with location parameters. The only requirement for tuple objects is to implement the interface `ITuple`, de£ned in a separate package providing a lightweight implementation of a tuple space. As for location parameters, LIME provides two classes, `AgentLocation` and `HostLocation`, which extend the common superclass

`Location` by enabling the de£nition of globally unique location identi£ers for hosts and agents. Objects of these classes are used to specify different scopes for LIME operations. Thus, 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 £nal destination, including also misplaced tuples. Note how, in the current implementation of LIME, probes are always restricted to a local subset of the federated tuple space, as de£ned by the location parameters. An unconstrained de£nition, like 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 is allowed to insert.

The remainder of the interface of `LimeTupleSpace` is devoted to managing reactions; other relevant classes for this task are shown in Figure 3. Reactions can either be of type `LocalizedReaction`, where the current and destination location restrict the scope of the operation, or `UbiquitousReaction`, that speci£es the whole federated tuple space as a target for matching. The type of reactions is used to enforce the proper constraints on the registration through type checking. These classes have the abstract superclass `Reaction` in common, which de£nes a number of accessors for the properties established on 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 speci£es the actions taken when the reaction £res, and a mode. The `ReactionListener` interface requires the implementation of a single method `reactsTo` that is invoked by the runtime support when the reaction actually £res. 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` and `ONCEPERTUPLE`, de£ned in `Reaction`. `ONCE` speci£es that the reaction is executed only once and then deregistered in the same atomic step. When `ONCEPERTUPLE` is speci£ed, the reaction remains registered but it never executes twice for the same tuple.

Reactions are added to the ITS by calling either `addStrongReaction` or `addWeakReaction`. Only

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

Figure 3. The classes required for the de£nition of reactions on the tuple space.

`LocalizedReaction` can be passed to the former, as prescribed by the LIME model. All the reactions passed as a parameter are registered in a single atomic step, i.e., processing of reactions takes place only after all reactions have been inserted in the `LimeTupleSpace`, and yet before any other operation takes place on it. The latter does not provide such a guarantee, as weak reactions could be spread on multiple hosts and thus enforcing the property above would entail a distributed transaction among all the hosts involved. Registration of a reaction in any case 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, or to register the same reaction with a strong and then subsequently with a weak semantics.

3. Design and Implementation of LIME

In this section we look behind the scenes of the LIME programmer interface, providing some insights about the internal structure of the `lime` package and of the associated run-time support. The presentation proceeds through increasing levels of complexity, £rst describing how the simple notion of a private, non-shared tuple space is made available through the `LimeTupleSpace` class. We then describe the components that enable the local transient sharing that determines a host-level tuple space and £nally we show how the illusion of a federated tuple space enabling transient sharing across remote hosts is provided.

Private Tuple Space. A private tuple space simply provides a Linda tuple space which is permanently attached to an agent. The agent enjoys exclusive access to the tuple space and can leverage off the strong reaction feature of LIME. Furthermore, since the private tuple space can later be shared, support for operations annotated with tuple location parameters is also provided. This core functionality is supported by two objects that belong to every `LimeTupleSpace`: `ITupleSpace` and `Reactor`. The £rst provides all standard Linda operations while the second manages the registered reactions as well as the execution of the reactive program.

One of the early decisions in the design of LIME addressed the implementation of the core tuple space support. Analysis of available systems revealed that they provide a rich set of features with large variations in terms of expressiveness, performance, and often semantics. The need for a simple, lightweight implementation, combined with the desire to provide support and interoperability with industrial-strength products, led us to the development of an adaptation layer that hides from the rest of the LIME implementation the nature of the underlying tuple space engine. This layer is provided by a separate package called LIGHTS, developed by one of the authors. `ITupleSpace`, together with the already mentioned `ITuple`, and `IField`, are the interfaces that provide access to the core tuple space functionality. Adapter classes implementing these interfaces are loaded at startup to translate operations into those of the supported tuple space engines. Currently, adapters exist for our lightweight tuple space and for IBM's TSpaces [3].

To support tuple locations, a design decision was made to explicitly augment each tuple within LIME with two location £elds, but, at the same time, to hide this internal representation from the application programmer. The user maintains access to and control over the location information only through extended `LimeTupleSpace` operations.

The other key component of the `LimeTupleSpace` is the `Reactor` object which contains the list of registered reactions which make up the reactive program. The current implementation supports reactions to changes in state and not to the mere occurrence of an operation. This means that execution of the reactive program must be triggered only when the content of the tuple space changes, i.e., as part of the execution of the `out` method of the `LimeTupleSpace`. To evaluate the reactive program to £xed point after every such change, the list of reactions is cycled through in a round robin fashion until no reaction is enabled. To avoid evaluating a reaction needlessly over the same tuples on each iteration of the reactive program, our `Reactor` adopts an optimized strategy that, during execution of the reactive program, separates the tuples written to the tuple space as a consequence of the £ring of a reaction from those that have already been checked, thus avoiding looking at the same tuple more than once per evaluation of a reaction. While this complicates the management of the tuple space during the evaluation of reactions, this concern is outweighed by its advantages, especially during the processing of `ONCEPERTUPLE` reactions which, as will be seen in Section 4, are a major asset during development.

Host-Level Tuple Space. Transient sharing of a `LimeTupleSpace` object is under the explicit control of the agent that creates it. Once sharing is turned on, a host-level tuple space is created. In order to properly enforce the semantics of transient sharing and to take into account engagement and disengagement of local tuple spaces, implementation of the host-level tuple space abstraction requires host-wide management of tuple space access. This management is provided by instances of the class `LimeTSMgr`. At run-time, a `LimeTSMgr` object is created when the £rst `LimeTupleSpace` instance with a given name is engaged. Subsequent engagements of `LimeTupleSpace` objects with the same name refer to the same `LimeTSMgr`.

Upon local engagement of a given tuple space, the `LimeTupleSpace` object surrenders the control of its own `ITupleSpace` object. Thus, the implementation of the methods providing access to the tuple space no longer operate directly on the `ITupleSpace`. Instead, operation requests are forwarded to the corresponding `LimeTSMgr`, and the calling agent is suspended, waiting for the result. Operation requests are queued and serially executed at the `LimeTSMgr`, which runs in a separate thread of control. This way, synchronization among concurrent accesses performed through different `LimeTupleSpace` instances is obtained structurally, by con£ning all tuple space accesses to a synchronized queue.

In the current implementation during engagement, not only does the `LimeTupleSpace` surrender control of its tuple space, but the content of the `ITupleSpace` object is physically merged into another `ITupleSpace` object associated with the `LimeTSMgr`. This latter object is a concrete representation of the host-level tuple space. Similarly, the reactive statements of each `LimeTupleSpace` instance are moved into a `Reactor` object associated with the `LimeTSMgr`. This design choice optimizes for tuple queries, and due to the movement of data this solution is most appropriate when con£guration changes are infrequent. Experiences with logical mobility applications have led us to consider the alternate solution of keeping tuples and reactions associated with the `LimeTupleSpace`, and allowing the `LimeTSMgr` to reference them. Future plans include extending our run-time support to allow the application designer to choose the most appropriate scheme.

In contrast to the private `LimeTupleSpace`, blocking operations are enabled on shared tuple spaces. In the case where a matching tuple is found immediately, no special processing is necessary and the `LimeTSMgr` releases the agent with the appropriate result (identical to the processing of a non-blocking operation). If no matching tuple exists, a mechanism is needed to detect when a matching tuple shows up, and to notify and release the waiting agent. The realization that this kind of processing is somehow reactive led to a solution that exploits the notion of reaction not only as part of the programming interface, but also as a core element of system design. Speci£cally, for each blocking operation that does not £nd immediately a matching tuple, a strong reaction with the speci£ed template is created with a system-de£ned `ReactionListener`. This listener is called just as any other LIME reaction listener, that is, with a `ReactionEvent` parameter containing the matching tuple that triggered the reaction. In the case of a **rd**, the listener returns a copy of the matching tuple to the suspended agent; in the case of an **in**, the matching tuple is £rst removed from the host, then returned to the waiting agent.

Federated Tuple Space. While the ultimate target environment for LIME is an ad hoc network where mobile hosts may move arbitrarily and mobile agents can roam among them, we recognize that such an ambitious task is likely to fail if not backed up by an initial evaluation of the primitives chosen. For this reason, our £rst version of LIME is based on a more constrained scenario that allowed us to quickly develop a £rst implementation and gather feedback from applications. We assume two basic properties of the underlying network. First, hosts announce their intentions to join and leave the LIME community of hosts. This allows us to control the engagement and disengagement processes without concerning ourselves with the possibilities of data loss due to sudden disconnection in the middle of a remote operation. Second, a LIME community is created by hosts joining and leaving *one at a time*. In other words, we do not yet support the engagement of two distinct LIME communities. Further, all hosts must be able to receive multicast and unicast messages from all other hosts in the community.

The management of system con£guration changes is key to moving from the host-level to the federated tuple space. To ensure that all hosts maintain consistent views of the community members, engagement and disengagement are implemented as community-wide transactions which are managed by a dynamically elected leader. A transaction is triggered by a multicast message from the initiator, to all members of the community. Upon receipt of this message, each host locally prepares for the transaction and informs the leader of its readiness. When all hosts are ready, the leader begins the transaction, initiating, for an engagement, the exchange of misplaced tuples and weak reactions, and, for both engagement and disengagement, the updating of the `LimeSystem` tuple space.

The mechanisms to support weak reactions in the federated environment exploit the strong reaction mechanism. When a weak reaction is registered, the programmer's `ReactionListener` object is stored by a `weakReactionMgr` object. A system-de£ned strong reaction is registered with the reactor of each of the hosts potentially involved in the weak reaction. When the strong reaction £res, either local or remote with respect to the subscriber, a message containing the `ReactionEvent` is passed to the `weakReactionMgr` and the user's `ReactionListener` is executed. In the case of a `ONCE` reaction, we must be careful to execute the `ReactionListener` only one time, even though multiple matching tuples may be returned from different hosts in the system.

The processing of remote, blocking tuple space operations exploits the weak reaction mechanism at the federated level much the way the blocking operations are handled at the host-level. A remote blocking **rd** is identical to a `ONCE`, weak reaction with a system de£ned `ReactionListener` which releases the blocked agent with the matched tuple. A remote blocking **in** is slightly more complex. First, a weak reaction is used to identify a host where a matching tuple exists. Second, an **inp** is performed on that host to attempt to retrieve the tuple. If the **inp** returns a tuple, the agent is released, but if no tuple is returned, the agent continues to wait.

Details about the Current Implementation. Communication is completely handled at the socket level, requiring no support for RMI or other communication mechanisms. The `lime` package is roughly 5,000 non-commented source statements, resulting in an approximately 100 Kbyte `jar` £le. The `lighTS` lightweight tuple space implementation and the adapter for integrating multiple tuple space engines adds an additional 20 Kbyte of `jar` £le.

4. Developing Mobile Applications with LIME

Application development is the last phase of our research strategy, and the one where the abstractions inspired by formal modeling and embodied in the middleware are evaluated against the real needs of practitioners. In this section we present two applications that exploit the current implementation of LIME in a setting where physical mobility of hosts is enabled. The £rst one involves the ability to perform collaborative tasks in the presence of disconnection, while the second one revolves around the ability to detect

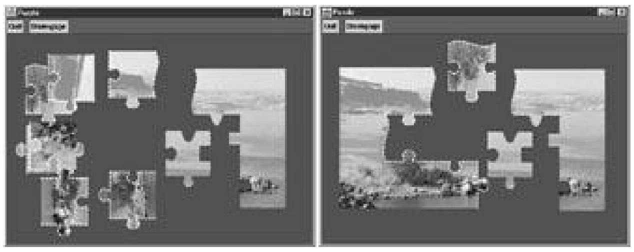

Figure 4. ROAMINGJIGSAW. The left image shows the view of a disconnected player which is able to assemble only pieces it selected. The right image shows the view after the player re-engages with the other players, seeing assemblies that occurred during disconnection.

changes in the system con£guration.

4.1. ROAMINGJIGSAW: Accessing Shared Data

Scenario. Our £rst application, ROAMINGJIGSAW as shown in Figure 4, is a multi-player jigsaw assembly game. A group of players cooperate on the solution of the jigsaw puzzle in a disconnected fashion. They construct assemblies independently, share intermediate results, and acquire pieces from each other when connected. Play begins with one player loading the puzzle pieces to a shared tuple space. Any connected player sees the puzzle pieces of the other connected players and can select pieces they wish to work with. When a piece is selected, all connected players observe this as a change in the colored border of the piece, and within the system, the piece itself is moved to be co-located with the selecting player. When a player disconnects, the workspace does not change, but the pieces that have been selected by the departing player can no longer be selected and manipulated. From the perspective of the disconnected player, pieces whose border is tagged with the player's color can be assembled into clusters. When a player reconnects, she becomes able to further redistribute the pieces, and to view the progress made by the other players with respect to any clusters formed since last connected.

This application is based on a pattern of interaction where the shared workspace provides an accurate image of the global state of connected players but only weakly consistent with the global state of the system as a whole. The user workspace contains the last known information about each puzzle piece.

ROAMINGJIGSAW is a simple game that exhibits the characteristics of a general class of applications in which data sharing is the key element. This design strategy may be adapted easily to any applications in which the data being shared may change, e.g., sections of a document in a collaborative editing application or paper submissions to be evaluated by a program committee.

Design and Implementation. The basic data element of ROAMINGJIGSAW is the puzzle piece. When a player selects a piece or joins together several pieces, a new tuple for the new piece is written and the old pieces are removed.

Critical operations are the detection of piece selection and assembly, the reconciliation on reconnection, and the engagement of a new player, all of which are handled by exploiting a single weak reaction with mode `ONCEPER-TUPLE` and type `UbiquitousReaction` whose scope is the whole federated tuple space. The reaction is registered for puzzle piece tuples and the reaction listener updates the user workspace with the information in the new puzzle piece, thus correctly maintaining the weakly consistent view of the workspace. Since the reaction is registered on the federated tuple space, the program receives updates about new descriptors without any need to be explicitly aware of the arrival and departure of players. Thus, the programming effort can focus just on handling data changes without worrying about the actual system con£guration.

Although all processing described thus far operates on the federated tuple space, £ne-grained control over the location of tuples is critical in dealing with disconnections. To ensure that a player has access to the pieces it selects even after a disconnection, piece selection actually moves the tuples into the local tuple space of the player. In addition, since we deal with a weakly consistent workspace, a player must be prevented from selecting a piece that is currently not present in the federated tuple space. For this reason the implementation uses the **inp** operation on the tuple space of the player last known to have the piece. If the piece is returned, it is properly moved to the local tuple space of the new owner, and the selection is successful. If no tuple is returned, it means that the piece is unavailable for selection.

4.2. REDROVER: Detecting Changes in Context

Scenario. Our second target application is a spatial game we refer to as REDROVER in which individuals equipped with small mobile devices form teams and interact in a physical environment augmented with virtual elements. This forces the participants to rely to a great extent on information provided by the mobile units and not solely on what is visible to the naked eye.

REDROVER is the initial step in the development of a suite of virtually augmented games to be carried out in the real physical world. Currently, the game is limited to seeking the physical ¤ag of another team and clustering around the player who £nds the ¤ag. Each player is equipped with a digital camera which can be used to share a snapshot of the current environment with team members who may be separated physically by walls or other barriers, but remain within radio communication range. Finally, players know

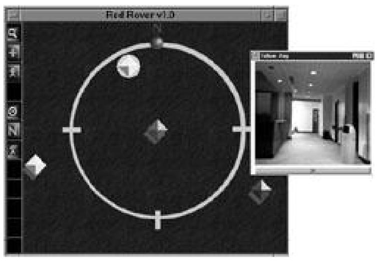

Figure 5. RedRover. The main console of RedRover, and the most recent camera image of a connected player.

and share their precise location in space so that all connected players can maintain an image of the playing £eld displaying the relative location of all participants

As with RoamingJigsaw, RedRover is a simple game but it has potential to be extended to real world scenarios such as the exploration of an unknown area by a group of people or robots.

Design and Implementation. The dominant feature of the user display is the current location of each connected player within the playing £eld. This is maintained in a strongly consistent manner, i.e., by displaying precisely the players which are connected and their most recent location update. Each time a player moves, a tuple representing its location is written to the federated tuple space. All players register a weak `ONCEPERTUPLE` ubiquitous reaction for these tuples and the screen is updated with each reaction.

To detect a player disconnection, we make use of the LimeSystem tuple space and register a reaction for the departure of a player (represented by a host). The listener of this reaction changes the connected status of the player and the user display is updated to replace the standard image of the player with a "ghost image" indicating that the player was once present, but is no longer connected. Similarly, when a host reconnects, a reaction on the LimeSystem tuple space £res to indicate its arrival and a query is done to the tuple space to determine its location.

An important feature of RedRover is the separation of data to be shared with teammates versus information available to all game players. For example, it is desirable to inform only team members of the ¤ag capture. Therefore, this information is written to a team-only tuple space, while general information, such as player location, is written to a separate game tuple space.

5. Discussion

In this section we discuss our research contributions with an emphasis on lessons learned from exploiting Lime in the mobile applications presented in the previous section. We also compare Lime to similar projects found in literature.

Re¤ections and Lessons Learned. The development of Lime is the result of a continuous interplay among the definition of the underlying formal model, the design and implementation of the middleware, and its evaluation on mobile applications. The development of a model for Lime, and its formalization, favored a better understanding of the abstractions provided by the middleware. In particular, by keeping the programming interface as close as possible to the operations de£ned in the formal model, we made it easy to communicate and reason about the functionality of the system and its use in applications. The ability to think about abstractions in a setting unconstrained by implementation details favored a style of investigation characterized by a more radical perspective, where the decisions driving the modeling and the de£nition of the main abstractions where mostly determined by the need for expressiveness and completeness.

This view was greatly re£ned when we started the design and implementation of the middleware. An example of re£nements that took place is provided by the notion of reaction which was motivated by the reactive statements of Mobile Unity, but which proved too strong to be reasonably implemented. Other re£nements were the result of unforeseen needs on the part of the application programmer, as was the case with the reaction mode. Speci£cally the `ONCEPERTUPLE` mode turned out to be an important mechanism in both RedRover and RoamingJigsaw.

Using Lime for application development made it possible for us to evaluate the usefulness of its programming abstractions and constructs. Our experiences corroborated the hypothesis that weak reactions on the federated tuple space provide the programmer with a highly effective construct that simpli£es programming. The execution of a single operation is suf£cient to guarantee future noti£cation of every event occurring over the whole federated tuple space, independently of changes in the con£guration. Interestingly, this power has a cost; the implementation of weak reactions is probably the most complicated portion of the current Lime software—this should be expected, since we are shifting a great deal of complexity away from the programmer and into the run-time support.

Another interesting byproduct of these empirical evaluations is an understanding of the programming and architectural styles fostered by Lime and recurring in mobile applications. Interestingly, in one of the applications its functionality must be provided *despite* mobility, while in the second case the functionality *exists because of* mobility. In this and other application typologies, a recurring dilemma is between an application style that provides a weakly consistent view of the system in the presence of mobility, and one that provides a fully consistent view that takes into account departure and arrival of mobile units. In our experience both styles are naturally accommodated by the abstraction of a transiently shared tuple space and use of the LimeSystem tuple space. Our "developers", mostly graduate and undergraduate students, found it easy not only to *program* applications with LIME but, most importantly, to *think* about the application in terms of the metaphors characteristic of the underlying LIME model.

Actually, the particular programming style induced by LIME, albeit biased by the limited range of applications considered thus far, is quite different from what we initially expected. This is especially true in the case of weak reactions and the LimeSystem tuple space. Reactive programming was not part of the initial core of LIME which was envisioned to be a coordination framework founded on the idea of transiently shared tuple spaces accessible exclusively through Linda operations. Similar circumstances surrounded the LimeSystem. It was initially thought of as an add-on to support very speci£c needs. Instead, these abstractions turned out to play a key role in the design of both ROAMINGJIGSAW and REDROVER. We already reported about the use of weak reactions and ONCEPERTUPLE and we noted that the LimeSystem tuple space provides full context awareness by exposing changes in the con£guration of the system.

Related Projects. LIME is not alone in its exploitation of the decoupled nature of tuple spaces for the coordination of mobile components. The Limbo platform [1] for quality of service communication among mobile hosts offers a *universal tuple space* which registers all tuple spaces, a notion similar to the LimeSystem tuple space. The TuCSon coordination model [6] for logically mobile Internet agents provides a kind of transparent access to shared context by giving programmers the ability to reference a tuple space based on a partial name which is resolved to a local version of the tuple space.

It is interesting to note how the notion of reaction put forth in LIME is profoundly different from similar event noti£cation mechanisms such as those provided by TuCSoN, TSpaces [3], and Javaspaces [4]. In these systems, the detected events are the actual operations performed on the tuple spaces, while in LIME, reactions £re based on the state of the tuple space itself. One other important difference is the power of the atomicity guarantees of the LIME strong reactions. For example, with a strong, local reaction, the execution of the listener is guaranteed to £re in the same state in which the matching tuple was found. No such guarantee can be given with an event model where the events are asynchronously delivered.

6. Conclusions

LIME is our £rst attempt at designing middleware for mobile systems based on the theme of coordination. The notion of transiently shared tuple spaces is part of a larger vision we refer to as *global virtual data structures*. This concept starts with the notion of a global, persistent, shared data structure accessible to all mobile agents but distributes it among mobile components and provides operations for sharing and manipulating the structure based on connectivity. While the choice of sharing Linda tuple spaces has proven useful, we anticipate applying this strategy to other kinds of data such as graphs or trees. The operations and semantics must be rede£ned, but the underlying notion of transient sharing based on connectivity remains. Finally, we are currently in the process of broadening our view of what is necessary for successful mobile middleware. Clearly, adaptability to different mobility scenarios is crucial and is something we have not explored fully within the con£nes of LIME. Experience to date has been instrumental in helping us develop a new strategy for structuring the LIME middleware and an effort is under way to achieve a multilayered modular design that can be adapted to mobile hosts of varying capabilities and to the construction of middleware based on a variety of coordination models.

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

Acknowledgments. This paper is based upon work supported in part by the National Science Foundation (NSF) under grant No. CCR-9624815. Any opinions, £ndings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily re¤ect the views of NSF.

References

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

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

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

[4] JavaSpaces. The JavaSpaces Speci£cation web page. `http://www.sun.com/jini/specs/js-spec.html`.

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

[6] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *Proc. of the 1999 ACM Symp. on Applied Computing (SAC'00)*, February 1999.

[7] G. Picco, A. 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.