LIME: A Middleware for Physical and Logical Mobility

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, report about its initial evaluation in applications that involve physical mobility, and discuss lessons learned and future enhancements that will drive its evolution.

1 Introduction

Middleware has emerged as a new development tool which can provide programmers with the benefits 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-specific languages and systems. The approach is intellectually attractive and economical at the same time. For the programmer, middleware offers a clean model that can be easily understood and readily adopted without the need to acquire a new set of programming skills or to delve into the intricacies of a sophisticated formal model. For the software engineer, middleware provides a vehicle by which new concepts and design strategies may be packaged and disseminated without the high cost associated with complex tool sets and compilers. For these reasons, middleware is enjoying growing popularity in the distributed computing arena. Given the complexities associated with software involving mobile hosts and agents, middleware is expected to establish itself as an important new technology in mobility as well. This paper is about middleware for mobility, 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 contents are controlled by the connectivity among mobile hosts. Individual programs perceive the effects of mobility as behind-the-scene changes in the contents 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. The temporal and spatial decoupling that made Linda an effective tool for parallel programming is preserved while accommodating the distinct nature of mobile computing.

When viewed in the broader context of mobility, LIME is indeed a new breed of middleware. Earlier work treated logical mobility essentially as a new *design* tool for the developers of distributed applications. The ability to reconfigure dynamically the binding between hosts and application components provides additional flexibility and, under given conditions, improved bandwidth utilization. On the other hand, physical mobility was viewed as a source of new *requirements* for distributed applications, by defining a very challenging target execution environment. These different roles are mirrored in the characteristics of the corresponding middleware. Middleware for logical mobility focused on new abstractions that enable code and state relocation, whereas middleware for physical mobility often tends to minimize differences with respect to non-mobile middleware, by relegating, as much as possible, the differences into the underlying runtime support. Moreover, middleware for physical mobility has been application centered. For instance, the Bayou [8] system provided the core functionality needed to build database applications that can handle disconnection through reconciliation and data hoarding, largely hiding the mobility from the application programmer. In contrast with earlier work, LIME is general purpose, model-centric, and inclusive of both physical and logical mobility. It provides novel programming constructs or novel uses of established ones in the tradition of logical mobility but in a manner that is sensitive to the constraints imposed by the realities of physical 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 and plans for future enhancements (Section 5) followed by conclusions (Section 6) and references.

2 LIME: Linda in a Mobile Environment

The LIME model [1] aims at identifying a coordination layer that can be exploited successfully for designing applications that exhibit either logical or physical mobility. LIME borrows and adapts the communication model made popular by Linda [3]. In the remainder of this section we review the fundamental concepts of Linda, discuss how they are reshaped in LIME for use in the mobile environment, and finally present the programming interface that allows development of mobile applications with our current implementation of LIME.

2.1 Linda

In Linda, processes communicate through a shared *tuple space* that acts as a repository of elementary data structures—the *tuples*. A tuple space is a multiset of tuples that can be accessed concurrently by several processes. Each tuple is a list 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 on it. Tuples can be removed from a tuple space by executing **in**(p). Tuples are anonymous, thus their selection takes place through pattern matching on the tuple contents. The argument p is often called a *template*, and its fields contain either *actuals* or *formals*. Actuals are values; the parameters of the previous tuple are all actuals, while the last two parameters of ⟨"foo", ?integer, ?long⟩ are formals. Formals are like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by **in** is selected non-deterministically and without being subject to any fairness constraint. Tuples can also be read from the tuple space using the **rd** operation. Both **in** and **rd** are blocking. 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].

2.2 The LIME model

Linda characteristics happen to 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.

The only challenge to this conceptual vision comes from consideration tied to an efficient implementation, that must take into account issues like reducing latency or providing fault-tolerance, like in [9]. On the other hand, the view fostered by mobile computing is profoundly different, even at the conceptual level. When mobility is fully exploited, like in the case ad hoc networking, there is no predefined, static, global context for the computation. Rather, the global context is defined by the transient community of mobile units that are currently present, to which each unit is contributing with its own individual context. Since these communities are dynamically changing according to connectivity and migration, the context changes as well. This observation alone leads to the model underlying LIME that, although still based on the Linda notion of a tuple space, exploits it in a radically different way.

The Core Idea: Transiently Shared Tuple Spaces 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 in 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 represent the only context accessible to the unit when it is alone. Access to the ITS takes place using the Linda primitives mentioned in the previous section, 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 of the ITS changes dynamically in reaction to changes in the set of co-located mobile units.

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

Upon arrival of a new mobile unit, the content perceived by each mobile unit through its ITS is recomputed by taking into account the context of the new unit, in order to establish transient sharing. The tuples in the ITS of the new unit are merged with the current content of the shared tuple space, and the result is made accessible within the ITS of all the units currently co-located. This sequence of operations is called *engagement* of the tuple spaces, and is performed as a single atomic transaction. Similar considerations hold for the departure of a mobile unit, resulting in the *disengagement* of the corresponding tuple space. The content of the unit's tuple space is removed atomically from the transiently shared tuple space perceived by the remaining units through their ITS.

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. The notion of transiently shared tuple space is a natural adaptation of the Linda tuple space to a mobile environment. When physical mobility is involved, there is no place to store a persistent tuple space. Connection among machines comes and goes and the tuple space must be partitioned in some way. 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. In a sense, LIME takes the notion of decoupling proposed by Linda further, by effectively decoupling the mobile units from the global tuple space used for coordination.

Encompassing Physical and Logical Mobility Thus far, we glossed over the nature of the mobile unit at hand, that is, we never specified whether we talked about a mobile agent moving in logical space or a mobile host roaming the physical space. This is precisely because we believe the LIME notion of a transiently shared tuple space is applicable to a generic mobile unit regardless of its nature, as long as a notion of connectivity ruling engagement and disengagement is properly defined.

In LIME, mobile hosts are connected when a communication link is *available* between them. Availability may depend on a variety of factors, including quality of service, security considerations, or connection cost; they can all 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 link. Connectivity among mobile agents may depend also

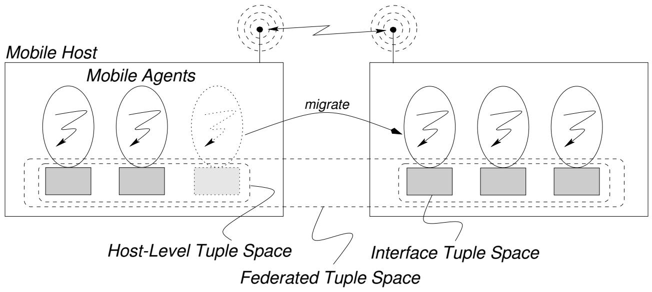

Figure 1: Transiently shared tuple spaces encompass physical and logical mobility.

on arrival and departure of agents, with creation and termination of mobile agents being regarded as a special case of connection and disconnection, respectively. Figure 1 depicts the model adopted by LIME.

In LIME, mobile agents are the only active components in the system; mobile hosts are mainly roaming containers for agents, to which they provide connectivity and execution support. Thus, in other words, mobile agents are the only components that carry a "concrete" tuple space along with them.

Co-location of mobile agents determines a *host-level tuple space* that is transiently shared among all such agents and accessible through each agent's ITS. As evident in Figure 1, the host-level tuple space can be regarded as the ITS of a mobile host, as it is permanently associated with it; if no mobile agents are currently hosted, the host-level tuple space is simply empty. Hence, transient sharing as we described for the ITS of mobile agents can be applied to the host-level tuple spaces. Hosts that are connected merge their host-level tuple spaces into a *federated tuple space* whose content is transiently shared across the network. When a federated tuple space is established, access to 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 to an agent residing on some remote 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.

Degrees of 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 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 a more fine-grained control upon the portion of the context that need to be accessed. For instance, the application may require control over the agent responsible for holding a given tuple, and this cannot be specified only in terms of the global context. Also, performance and efficiency considerations may come into play, like in the case where application information would enable access aimed at a specific host-level tuple space, thus avoiding an expensive query on the whole federated tuple space.

This fine-grained control over the context is provided in LIME by extending Linda operations with tuple location parameters that allow to operate on different projections of the transiently shared tuple space. Tuple locations parameters are expressed in terms of agent identifiers or host identifiers, as these identify the scope for the transiently tuple space that holds the tuple, i.e., the agent's tuple space and the host-level tuple space, respectively.

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**$[\lambda]$ 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, i.e., either co-located or located on a connected mobile host, the tuple t is moved to the destination location. The combination of the two actions—the insertion of the tuple in the ITS of ω and its instantaneous migration to the ITS of λ—are performed as a single atomic operation. On the other hand, if λ is not currently connected, the tuple remains at the current location, the tuple space of ω. This "misplaced" tuple, if not withdrawn[2], will remain such unless λ becomes connected. In this case, the tuple, which is nonetheless accessible through the ITS independently of its current location, will migrate to the tuple space associated with λ as part of the atomic sequence of operations performed during engagement. Hence, using **out**$[\lambda]$, the caller can specify that the tuple, albeit shared, is supposed to be placed within the tuple space of agent λ. This way, the default shared 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 come into play also to provide variants of the **in** and **rd** operations that allow access to a slice of the global context. In LIME, these operations are annotated as **in**$[\omega, \lambda]$ and **rd**$[\omega, \lambda]$, where the current and destination locations defined earlier are used. More details are provided at the end of this section, when we review the programming interface embodying the LIME model.

Disengagement relies on the notion of tuple location, as well. Upon occurrence of a disconnection, be it the departure of an agent or a broken communication link, the transiently shared tuple space is partitioned into its constituents, i.e., as if each mobile agent were alone. In this situation, 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. Then, these ITS are merged back according to the new configuration of the system after disconnection, effectively creating two partitioned transiently shared tuple spaces. It is interesting to note, however, that the above is just a conceptual representation of the disengagement process. In practice, no tuple transfer is needed to comply with the above process, provided the atomicity of engagement and of the **out**$[\lambda]$ operation are 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)configuration that generated those changes. For instance, if the probe **inp**$[\omega, \lambda](p)$, looking in the tuple space of agent ω for tuples matching p and destined for agent λ, fails, this simply means that no tuple matching p is available in the current projection over the location parameters $[\omega, \lambda]$ of the federated tuple space. No information is available to determine whether the failure is determined by the fact that agent ω does not have those tuples in its tuple space, or whether agent ω is not part of the community at the moment.

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 only with application data. Although this perspective is often enough for the requirements of many mobile applications, in some cases the configuration 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 provides this form of awareness of the system configuration using the same set of abstractions discussed thus far. Information about the configuration of the system can be accessed through a transiently shared

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

tuple space. Tuples contain information about the mobile components present in the community, and their relationship, e.g., which tuple spaces they are sharing or, for mobile agents, which host they reside on. This tuple space, conventionally named LimeSystem, has the peculiarity of providing only read-only access: thus, only **rd** operations are allowed. Furthermore, reactions can be set on the tuple space, to enable actions to be taken in response to a change in the configuration of the system. All agents are permanently bound to LimeSystem. Thus, transiently shared tuple spaces, including the LimeSystem tuple space and the ITSs defined for application purposes, enable the definition of a fully context aware style of computing.

Reacting to Changes in Context Mobility enables a highly dynamic environment, where reaction to changes constitutes a major fraction of the application design. At a 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 shows up in the tuple space. Nevertheless, in practice this solution has a number of drawbacks that are well-known in literature, and are a consequence of the different perspective adopted by Linda, which expects agents to poll proactively and synchronously the context for new events, rather than specify the actions to be executed reactively and asynchronously upon occurrence of an event.

LIME extends tuple spaces with a 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 is based on Mobile UNITY reactive statements, described in [6]. After each operation on the tuple space, a reaction is selected nondeterministically and the pattern p is matched against the content of the tuple space. If a matching tuple is found, s is executed, otherwise the reaction is equivalent to a no-operation. This selection and execution proceeds until there are no reactions enabled, and normal processing of tuple space operations can resume. Thus, reactions are executed as if they belonged to a separate reactive program that is run to fixed point after each non-reactive statement. Blocking operations are not allowed in s, as they would conflict with the semantics of the processing of reactions, which must reach termination before standard processing is resumed. These semantics offer an adequate level of reactivity, because all the reactions registered are executed before any other statement of the co-located agents, including the migration statements. Thus, the programmer's effort in dealing with events is minimized.

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

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

2.3 Programming with LIME

We conclude the presentation of the LIME model by briefly commenting upon the programming interface that is currently provided in the implementation of LIME we report about in this work.

The class `LimeTupleSpace`, shown in Figure 2[3], embodies the concept of a transiently shared tuple space. Objects of this class are created by specifying an instance of the `Agent` class, which essentially provides a means to uniquely identify a mobile agent. The thread associated to such agent object will be the only one allowed to perform operations on the `LimeTupleSpace` object; accesses by other threads will 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 second parameter for the constructor of `LimeTupleSpace`. The name determines the sharing rule; only tuple spaces with the same name are transiently shared. For instance, this enables an agent to exchange information with a service broker about the available CD resellers by transiently sharing the corresponding ITS, and

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

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

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

then subsequently share information about a given title and the payment options with the reseller selected through a different ITS, thus keeping separate the information concerned with different tasks and different roles.

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 turn out to be useful just as a primitive data structure for local data storage. As a matter of fact, all tuple spaces are initially created as 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 or vice versa. Calling this method effectively triggers engagement or disengagement of the corresponding tuple space. Sharing properties can also be enabled in a single atomic step for multiple tuple spaces owned by the same agent by using the class method `setShared`.

`LimeTupleSpace` contains also the Linda operations needed to access the tuple space, as well as their variants annotated with location parameters. Tuple objects must implement the interface `ITuple`, defined in a separate package that provides a definition for a Linda tuple space that is independent on the actual runtime support used. As for location parameters, LIME provides two classes, `AgentLocation` and `HostLocation`, which extend the common superclass `Location` by 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. 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 an unspecified location parameter. Thus, for instance, `in(cur,Location.UNSPECIFIED,t)` returns a tuple contained in the tuple space of `cur`, regardless of its final destination, thus including also misplaced tuples. Note how typing rules allow to constrain properly the nature of the current and destination location according to LIME rules. Thus, for instance, the `destination` parameter is always an `AgentLocation` object, as agents are the only carriers of a "concrete" tuple space in LIME. Specifying a `HostLocation` as a destination for a tuple would result in the impossibility to assign a responsible for the tuple when the host-level tuple space becomes partitioned due to disengagement. Note also how, in the current implementation of LIME, probe are always restricted to a subset of the federated tuple space, as defined by the location parameters. An unconstrained definition, like the one provided for `in` and `rd`, would involve a distributed transaction in order to preserve the semantics of the probe across the whole transiently shared 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 nonetheless permanently and exclusively associated to an agent, the execution of a blocking operation would immediately suspend the agent forever, waiting for tuples that no other agent is allowed to insert. In this case, a run-time exception is thrown instead.

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 tuple space scanned for matching, or `UbiquitousReaction`, that specify the whole federated tuple space as a target for matching. The type of reactions is used to enforce the proper constraints on the registration of reactions through type checking. These classes have the abstract superclass `Reaction` in common, which defines a number of accessors for the properties set 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 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 runtime 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` and `ONCEPERTUPLE`, defined in `Reaction`. `ONCE` specifies that the reaction is executed only once and then deregistered automatically in the same atomic step. When `ONCEPERTUPLE` is specified instead, 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 `LocalizedReaction` can be passed to the former, as prescribed by the LIME model. Due to the different semantics, this operation has different atomicity guarantees. The former guarantees that 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 guarantee, as weak reactions could be spread on multiple hosts and thus enforcing the property above would entail a distributed transaction among all the nodes involved.

Registration of a reaction in any case returns an object `RegisteredReaction`, that can be used to deregister a reaction with the method `removeReaction`. `RegisteredReaction` basically acts as a "ticket stub" for the registration of the reaction, and provides additional information about the registration process. The decoupling between the reaction used for the registration and the `RegisteredReaction` object returned allows for

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

Figure 3: The classes `Reaction`, `RegisteredReaction`, `ReactionEvent`, and the interface `ReactionListener`, required for the definition of reactions on the tuple space.

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, by providing some insights about the internal structure of the `lime` package and of the associated run-time support. The presentation will proceed through increasing levels of complexity. We first describe how the simple notion of a private, non-shared tuple space is made available through the `LimeTupleSpace` class. Then, we move on to describe the components that enable the local transient sharing that determines a host-level tuple space. Finally, we show how the illusion of a federated tuple space enabling transient sharing across remote nodes is provided.

Private Tuple Space A private tuple space essentially provides a Linda tuple space that is permanently attached to an agent. It 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 engaged through transient sharing, support for operations annotated with tuple location parameters must be provided as well.

The core functionality above is supported by two objects that belong to every `LimeTupleSpace` object. The first object has type `ITupleSpace` and provides exactly the functionality of a plain Linda tuple space, including blocking operations. The second object is of type `Reactor` and is in charge of running the reactive program constituted by the reactions registered on the `LimeTupleSpace` object after each operation.

When designing LIME, we had to face the decision about how to implement the core tuple space support. Analysis of available systems revealed that they provide a very rich set of features, with big variations in terms of expressiveness, performance, and often also semantics. The need for a simple, lightweight implementation, combined with the desire to provide support and interoperability with industry-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 this interfaces can be loaded at startup time to translate these operations into those of the tuple space engines supported.

Currently, adapters are in place for our lightweight tuple space implementation and for IBM's TSpaces [4]. Short term activities include the development of an adapter for Sun's JavaSpaces [5].

Support for operations annotated with tuple locations relies on the `ITupleSpace` object, although a change in the format of tuples is performed along the way. In fact, the design choice we made was to represent tuple location parameters as tuple fields, in order to simplify the implementation of the corresponding extended operations and, as we will see later, to simplify the retrieval of misplaced tuples during engagement. Nevertheless, this representation is hidden from the programmer, who is prevented from tampering directly with the location fields (which would possibly lead to changes in the semantics) and can refer to location fields only through the corresponding parameters in the operations provided by `LimeTupleSpace`. Thus, upon insertion, a tuple specified by the programmer is augmented with two location fields representing the current and destination location. These fields are then stripped down when operations accessing the tuple space, like **rd**, are performed. As it will be discussed in the remainder of this section, a third field containing a globally unique tuple identifier is also added, and it is used exclusively to support reactions with a `ONCEPERTUPLE` mode.

The `Reactor` object is the other key component of the `LimeTupleSpace`. It contains the list of registered reactions forming the reactive program, which gets changed through the methods of the `LimeTupleSpace` that add and remove reactions. The current implementation supports only 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 contents of the tuple space changes, i.e., as part of the execution of the `out` method of the `LimeTupleSpace`. The requirement for the reactive program to run to fixed point after every such change is achieved by cycling through the whole list of reactions in a round robin fashion until no reaction is enabled to fire. A straightforward implementation of this processing would probe the whole tuple space for a tuple matching the reaction's template every time a reaction is evaluated. Clearly, this would be quite highly inefficient even for a small number of reactions and tuples, especially in the case of reaction listeners that insert tuples of their own. For this reason, our `Reactor` adopts an optimized strategy that mirrors and separates, during execution of the reactive program, the tuples written to the tuple space as a consequence of the firing 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. This complicates the management of the tuple space during the evaluation of reactions because it must be kept consistent with the `Reactor`'s view. Nevertheless, in our experience this added complexity is far outweighed by the advantages gained, especially during the processing of `ONCEPERTUPLE` reactions which, as we will discuss, represent a major asset during development.

Host-Level Tuple Space Transient sharing of `LimeTupleSpace` objects is under the explicit control of the respective agent. Once sharing is turned on, a host-level tuple space is created. Implementation of this abstraction requires a host-wide, centralized management of the access to the individual tuple space objects, in order to properly enforce the semantics of transient sharing and to take into account engagement and disengagement of local tuple spaces. This management is provided by instances of the class `LimeTSMgr`.

At run-time, there exist one `LimeTSMgr` per each named transiently shared tuple space currently active. A `LimeTSMgr` object is created as soon as the first `LimeTupleSpace` instance with a given name is engaged. Subsequent engagements of `LimeTupleSpace` objects with the same name will refer to the same `LimeTSMgr`. Engagements of objects with a different name will refer to a different `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 enqueued by the `LimeTSMgr`, which runs in a separate thread of control, and thus their execution is serialized. This way, synchronization among concurrent accesses performed through different `LimeTupleSpace` instances is obtained structurally, by confining concurrent accesses in a synchronized queue.

In our current implementation, not only the `LimeTupleSpace` surrenders control of its tuple space, but the contents of the `ITupleSpace` object are physically merged upon engagement in another `ITupleSpace` object associated with the `LimeTSMgr`. This latter object becomes then a concrete representation of the host-level tuple space. Similarly, the reactive statements of each `LimeTupleSpace` instance are all moved, upon engagement, into a `Reactor` object associated to the `LimeTSMgr`. The rationale for this design decision lied in the fact that this solution optimizes for tuple queries, especially if the underlying tuple space engine adopts indexing mechanisms to provide faster access to tuples, like in the case of TSpaces. Thus, this solution is appropriate in the case where changes in the configuration are not very frequent. However, experience with applications and the development of our own lightweight tuple space made us consider more carefully the alternate solution of keeping tuples and reactions in the objects associated with the

LimeTupleSpace, and simply allow the LimeTSMgr to reference them. This latter solution, by eliminating the transfer of tuples and reactions during engagement and disengagement, is likely to provide better performance in the case of frequent mobility. In the short term, we will extend our run-time support to let the choice of the more appropriate strategy to the designer, who will evaluate it against application needs.

In contrast with LimeTupleSpace objects that are still private, when sharing is enabled blocking operations are allowed as well because multiple agents can write tuples to the host level tuple space. In the case where a matching tuple is found, no special processing is necessary and the LimeTSMgr releases the agent with the appropriate result. However, if no matching tuple exists, a mechanism must be established to detect when the tuple shows up, and immediately to notify and release the waiting agent. The realization that this kind of processing is somehow reactive led us to a design solution that exploits the notion of reaction not only as part of the programming interface, but also as a core element of system design.

For each blocking operation that does not find immediately a matching tuple, a strong reaction with the specified template is created, together with a system-defined ReactionListener. This listener will be called as any other Lime reaction listener, that is, with a ReactionEvent parameter containing the matching tuple triggering the reaction. In the case of a **rd**, the listener will simply return a copy of the tuple in the ReactionEvent object to the suspended agent; in the case of an **in**, the listener will also first remove the matching tuple from the host. Note that, in this latter case, the listener is guaranteed that the tuple is still in the tuple space, because the reactive program runs as a single atomic step.

Federated Tuple Space Creating the illusion of a transiently shared federation of tuple spaces is the ultimate goal of the abstractions provided by Lime. This is accomplished by building upon the choices and mechanisms discussed thus far. While, the ultimate target environment for Lime is an ad hoc network where mobile hosts may move unconstrained and mobile agents can roam among them, we recognize that such a task of monumental proportions is likely to fail if not backed up by an initial evaluation of the primitives chosen. For this reason, our first version of Lime is based on a more constrained scenario that allows us to quickly develop a first implementation and gather feedback from applications, as discussed in the next two sections. We assume that mobile units announce explicitly their intentions to join and leave the Lime community, which determines the ability to control programmatically in Lime the engagement and disengagement process. Also, the scenario we assume involves a single community of mobile hosts, all in communication

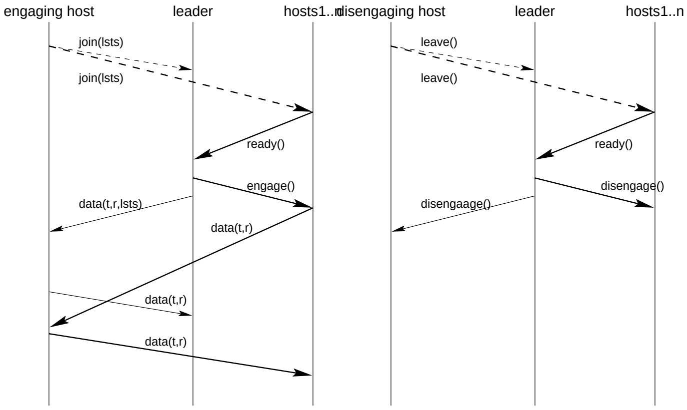

Figure 4: The engagement and disengagement protocols. Dashed lines indicate multicast messages, heavy solid lines represent multiple unicast, and regular lines are unicast messages. The data message from the leader may contain tuples (t), weak reactions (r), and all the tuples in the LimeSystem tuple space (lsts).

range. This latter constraint will change as we integrate an ad hoc routing protocol within our testbed network. Thus far, disengagement of a host always leaves the rest of the community connected, and hosts are able to join the community only one at a time, i.e., we do not yet support the engagement of two distinct Lime communities.

The management of changes in the configuration of the hosts is one of the key additions needed to move from the host-level to the federated tuple space. The engagement and disengagement protocols are implemented as community-wide transactions in order to maintain a consistent view among all hosts. To coordinate change requests in the configuration of the community and to ensure a total ordering of transactions among all hosts, the current version of our engagement protocol determines the presence of a leader in the community, with an election mechanism in place to deal with leader departure. The details of the protocols can be seen in Figure 4.

The first step of the protocol involves the engaging host, and presumes the availability of multicast support, which is exploited by the host to send a first message requesting the engagement. Upon receipt of the message, all hosts in the community prepare locally for engagement and inform the leader that they are ready. When the leader knows that all hosts are ready, the distributed transaction begins and the hosts in the community begin to exchange any misplaced tuples and new weak reactions with the new host.

One critical aspect of engagement is the update of the LimeSystem tuple space. This is accomplished in two steps. When a host first joins the Lime community, it sends a copy of the content of its own LimeSystem in the multicast message sent to all the hosts. This information is bound to contain only agents and tuple spaces present on that host, since the engaging host is not part of any other community. The information distributed to the members of the community serves to update their own LimeSystem in a way that is consistent with the configuration the system will assume after engagement is completed. In addition, when the leader sends its own tuple space information it will also send a copy of its LimeSystem tuple space to the new host. This way, the engaging host will be able to obtain a consistent view of the new configuration being built. Incidentally, this will also allow the engaging host to determine when it has exchanged data with all the members of the community, and thus it can resume regular processing. The disengagement protocol is similar albeit notably shorter than engagement, as there is no need to exchange data.

Besides changes in the configuration, the very task of enforcing the semantics of the operations we described in the previous section for the whole federated tuple space is complicated by distribution. In particular, much of the complexity actually lies in the mechanisms supporting weak reactions. These are based on the same idea exploited to handle blocking operations on the host-level tuple space. When a weak reaction is registered, the `ReactionListener` object specified by the programmer is inserted in a separate `weakReactionMgr` object, while a system-defined strong reaction is registered with the reactor associated with the `LimeTSMgr` of the hosts involved in the weak reaction. These strong reactions guard the host-level tuple space (or a single agent tuple space, depending on the value of its current location parameter). If the strong reaction is fired locally to the subscribing agent, the listener simply looks into the local `weakReactionMgr` and the user `ReactionListener` is executed. Alternately, if the reactor is remote, the listener sends a message to the subscriber's host with a `ReactionEvent`. When this message arrives, the user's `ReactionListener` is executed. In the case of a `ONCE` reaction, we must be careful to only execute the `ReactionListener` one time even though multiple matching tuples may be returned from different hosts in the system.

Federation also has an impact on remote processing of basic tuple space operations. Just as we were able to exploit the local reactor for the remote operations at the host level, here we utilize the weak reaction structure. A remote blocking **rd** is identical to a `ONCE`, weak reaction with a system defined `ReactionListener` which releases the blocked agent. A remote blocking **in** is slightly more complex as we may get responses back from multiple hosts, must return to that host to actually retrieve the tuple (using an **inp**) before releasing the agent.

Although the same `Reactor` serves both `ONCE` and `ONCEPERTUPLE` reactions, the processing of matching tuples are tailored based on this mode. After a `ONCE` reaction fires, the reaction is removed from the reactive program because the user's request has been satisfied. Alternately, a `ONCEPERTUPLE` reaction remains registered and must ensure that no single tuple causes the reaction to fire more than once. This is accomplished by keeping a list of the tuple identifiers which have already been reacted to within the `RegisteredReaction` itself. Each time a matching tuple is found, this data structure is queried and updated to determine if the `ReactionListener` should be executed. The implementation of the `Reactor` which separates newly written tuples from those which were in the tuple space prior to this round of the reactive program greatly improves the performance of `ONCEPERTUPLE` by not selecting a tuple more than once from a single local tuple space. However, because tuples can migrate and weak reactions can be uninstalled and reinstalled as connectivity changes, it is possible for a tuple to be selected more than one for a reaction, making the list of tuple identifiers necessary. By passing a relevant subset of the list of tuples already reacted to when an upon is reinstalled, additional duplication can be eliminated.

Details about the Current Implementation LIME is currently implemented completely in Java, with support for version 1.1 and higher. Communication is completely handled at the socket level—no support for RMI or other additional communication mechanisms is needed or exploited in LIME. The `lime` package is about 5,000 non-commented source statements, for about 100 Kbyte of `jar` file. The `lighTS` package providing a lightweight implementation of a tuple space and the adapter layer integrating multiple tuple space engine adds an additional 20 Kbyte of `jar` file. Thus far, it has been tested successfully on PCs running various versions of Windows and on hand-held devices running WindowsCE and using Lucent WaveLAN wireless technology.

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 two applications are typical of the physical mobility domain. The first one involves the ability to perform collaborative tasks in the presence of disconnection, while the second one revolves around the ability to detect changes in the system configuration. In each case, we present the corresponding application scenarios and a report about the way LIME has

Figure 5: 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 assembly that occurred during disconnection.

been exploited during development. The lessons learned from these experiences and the results of our empirical evaluation of Lime are presented in the next section.

4.1 RoamingJigsaw: Accessing Shared Data

Scenario Our first application, RoamingJigsaw as shown in Figure 5, 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. Additionally, the player can connect to other players 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. It is interesting to observe that the globally set goal of the distributed application, i.e., the solution of the puzzle, is built incrementally through successive updates to the local state, distributed to all other players either immediately if connected or in a "lazy" fashion if connectivity is not available at that time.

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

Design and Implementation The basic data element of RoamingJigsaw is the individual puzzle piece. For efficiency purposes, each piece is stored as a pair of tuples. The first contains the image of the piece which remains unchanged throughout the game. The second contains a descriptor that includes information about a piece or the cluster that includes it and the current owner of the cluster. When a player selects a piece or joins together several pieces, a new tuple with the updated information is inserted, and the old descriptor is removed.

The critical operations in the game are the detection of piece selection and clustering actions, the reconciliation on reconnection, and the engagement of a new player. All are handled by exploiting a single mechanism in Lime: a weak reaction with mode `ONCEPERTUPLE` and type `UbiquitousReaction`, its scope is the whole federated tuple space. The reaction is registered for tuples that match any cluster descriptor. The corresponding reaction listener updates the user workspace with the information in the matched descriptor and correctly maintains the weakly consistent view of the workspace. The amount of data transmitted for each update is minimized, because the reaction looks for descriptors and not for the individual piece images. However, in the case where a puzzle descriptor is received for a piece which the player never encountered before, as is the case during the engagement of a new player, the puzzle image is explicitly requested directly from the tuple space before the workspace is updated. 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 configuration.

Although all processing described so far has operated on the federated tuple space, fine-grained control over the location of tuples is critical in dealing with disconnections caused by mobility. When a player selects a piece to work with, the piece must remain part of the transiently shared tuple space, but its location is changed to that of the selecting player in order to enable it to disconnect without losing access to the descriptor. 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 these reasons, our implementation of RoamingJigsaw responds to an attempt to

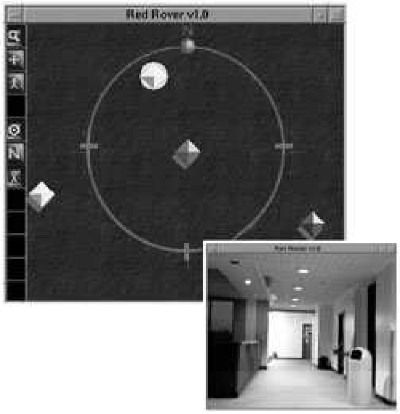

Figure 6: RedRover. The main console of RedRover, and the most recent camera image of a connected player.

select a piece by first performing an **inp** operation on the tuple space of the player last known to have the piece. If the piece is returned, it is properly rewritten 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 because the corresponding player is currently disconnected. This fact is communicated to the user by an audible beep.

We are presently developing a version of RoamingJigsaw that presents the user with a workspace that represents the current state of the system in a fully consistent way, i.e., only pieces belonging to users that are currently connected are seen through the workspace. All other pieces are removed from the workspace upon disconnection of the player that owns them and are redisplayed as soon as the player becomes connected again. In other words, the player sees exactly the pieces which are currently in the federated tuple space. This version may be easily coded by utilizing the `LimeSystemTupleSpace` to react to player arrivals and departures and by monitoring which puzzle pieces have been handed off to other players.

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. *BodyWare* will provide each player with global positioning system access, audio and video communication, range finding capabilities, and much more. For now, the game is limited to seeking and discovering the physical flag of the other team and clustering around the player who finds the flag. 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 and share their precise location in space so that all connected players can maintain an image of the playing field displaying the relative location of all participants As with RoamingJigsaw, RedRover is a simple game but it has great potential to be extended to real world scenarios such as the exploration of an unknown area by a group of people or robots. Our current efforts include the incorporation of a mapping mechanism which will allow users to define the elements of a region and share these results as they meet other users. Finally, the current implementation employs an artificial notion of player location, however, this can be trivially replaced with a global positioning system.

Design and Implementation in Lime The dominant feature of the user display is the current location of each connected player within the playing field. 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 when a player disconnects, 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. Upon reconnection, once the player moves, the `ONCEPERTUPLE` reaction gets the new location for the player. However, if the player stays put, the reaction on the location tuple will not fire. Nevertheless we can still update the player's status to connected by registering a reaction on the LimeSystem tuple space for the arrival of a player (host).

To handle the notification of the flag capture, each player can register a `ONCE` or `ONCEPERTUPLE` upon on the federated tuple space. When a player finds the flag it writes a tuple to the tuple space indicating this fact, and all registered players receive notification in the form of a dialog box indicating which player has the flag. To facilitate clustering around this player, it is useful to request the camera image of the player in order to identify obstacles not visible on the screen which must be maneuvered around. Because the image is requested from a specific player, we simply use the **rdp** operation rather than incurring of the overhead of the reaction.

Another feature of the implementation 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 flag capture. Therefore, this information is written to a team-only tuple space, while general information. such as player location is written to a game tuple space. The ability to have selective sharing of tuple spaces is an important feature and the first step towards introducing security considerations in LIME.

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 and report about future work and enhancements we plan for our middleware.

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

This view was greatly refined when we started the design and implementation of the middleware. An example of refinements that took place is provided by the notion of reaction. Reactions were motivated by an intuition of the importance of reacting to events in a mobile environment and were inspired by the notion of reactive statements in Mobile UNITY. Nevertheless, reactions as defined in Mobile UNITY imposed atomicity requirements that are in general too strong to be practical in a distributed setting. This consideration led to the notion of a weak reaction, which represents a seemingly reasonable compromise between the loose guarantees provided by common event mechanisms like those found in TSpaces and JavaSpaces and the full atomicity guarantees of strong reactions.

Other refinements were the result of unforeseen needs on the part of the application programmer. This was the case with the reaction mode. In the reactive model of Mobile UNITY, reactions are permanently enabled and it is up to the designer to specify the conditions under which they become disabled. Nevertheless, programming practice with the LIME model showed early on that some sort of automatic disabling of reactions is needed. In particular, the `ONCEPERTUPLE` mode turned out to be an important mechanism in developing both applications discussed in this paper.

The feedback coming from applications was not limited to the discovery of new primitives. The use of LIME made it possible for us to evaluate the usefulness of its programming abstractions and constructs. Experience with ROAMINGJIGSAW and REDROVER corroborated our hypothesis that the ability to register weak reactions on the whole transiently shared tuple space provides the programmer with highly effective constructs that simplify the programming task. The execution of a single operation is sufficient to guarantee future notification of every event occurring over the whole federated tuple space, independently of changes in the configuration. 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. A possible distinction can be made between applications whose main requirement is to enable sharing of data despite mobility and those where most of the computation is driven by reactions to changes in context, as is the case with the two applications we presented here. Interestingly, in one case the functionality of the application 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. Choosing one representation style or the other has non-trivial implications on the complexity of the overall design and development task, and on the primitives that must be used. If weak consistency is enough, the view can be built incrementally by exploiting the notification mechanism provided by weak reactions, usually in the

ONCEPERTUPLE mode. If, instead, a fully consistent view is required, additional, application specific machinery must be added in addition to using the LimeSystem tuple space to react (immediately) to changes in the system configuration. In our experience both styles are naturally accommodated by the abstraction of a transiently shared tuple space. 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 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 specific 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 configuration of the system. Although we initially thought this explicit knowledge could be bypassed by the observation of changes in the data context, experience with our applications (especially with REDROVER), showed that this hypothesis does not hold in general. The developer must resort to the LimeSystem tuple space. This causes no difficulties since the LimeSystem tuple space is perceived by the user as just another transiently shared tuple space with a different name and restricted access.

Finally, an issue that deserves careful evaluation is the extent to which the programmer is induced to duplicate the data present in the tuple space into some other runtime data structure, for performance reasons. For instance, in ROAMINGJIGSAW the re-paint of the workspace would involve retrieving from the federated tuple space all the pieces present at that moment. Clearly that is impractical, and the content of the tuple space is mirrored in a data structure that is kept consistent as changes are notified through reactions. In ROAMINGJIGSAW, such a mirroring is necessary in order to preserve weak consistency of the workspace, and to keep track of pieces that are no longer available. Nevertheless, this issue has more profound implications that have to do with the way the tuple space is actually used (i.e., as a coordination means or as a data repository). A comparative evaluation of the LIME programming style relative to the programming style induced by other middleware based on tuple spaces, like TSpaces [4] or JavaSpaces [5] remains to be carried out in the future.

5.2 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 [2] builds the notion of a quality of service aware tuple space which resides on mobile hosts. The quality of service information itself is stored in the tuple spaces and can be made accessible to agents on remote hosts. There is no notion of sharing data among the tuple spaces, however a *bridge agent* can be built which has references to multiple tuple spaces and can monitor and copy information among the spaces. Agents must also know explicitly which tuple space they wish to connect to. A *universal tuple space* exists which registers all tuple spaces and can be used to locate a space. This notion is similar to the LimeSystem tuple space. Limbo does not provide any mechanisms beyond the regular Linda operations to react to changes in the tuple space.

In contrast, the main focus in the TuCSoN coordination model [7] is a reactive mechanism which is used to create *programmable tuple spaces* which respond to the queries of mobile agents. When an agent poses a query to the tuple space, the registered event which matches the operation and template fires, and an action is atomically performed. Another feature of TuCSoN is the ability to either fully qualify a tuple space name, identifying the specific host where the tuple space relies, or providing a partial name and gaining access to a local version of the tuple space. There is no coordination between tuple spaces, and mobile agents only have access to the tuple spaces fixed at the hosts.

It is interesting to note how the notion of reaction put forth in LIME is profoundly different from similar extensions that allow notification of events in the tuple space, such as those provided by TuCSoN, TSpaces [4], and Javaspaces [5]. In these systems, the events that are detected are the actual operations performed by the accessing processes, while in LIME, reactions fire based on the state of the tuple space itself. One common application task we discovered early is the need to look for a tuple, and, if it is not present, and then wait for its appearance. Without transactions, this is complicated by the possibility for the tuple to be written in between the initial query and the installation of the event listener. However, transactions are complex and expensive. In LIME, a single reaction accomplishes the desired task. Furthermore, the atomicity guarantees of the local reactions are relatively powerful. For example, with a localized reaction, the execution of the listener is guaranteed to fire in the same state in which the matching tuple was found. No such guarantee can be given with an event model where the events are asynchronously delivered.

5.3 Future Directions for LIME

An important feature of the LIME model is the integration of physical and logical mobility. The ability of a mobile agent to move among mobile hosts opportunistically as connectivity is available makes the integration of the two models powerful. Our current design is already centered around the concept of an agent, which has the ability to create, access, and share `LimeTupleSpace` objects, however we do not yet allow for this agent to migrate. To accomplish this, current plans include integrating the μCODE toolkit into LIME. This will involve minimal modifications to the transaction protocols, as well as modifications to the `LimeServer` to handle the departure of agents.

A reality of the physically mobile environment is the unanticipated loss of a communication channel when a host moves out of range. This can cause an interruption of the engagement or disengagement transactions or data instability when a connection is lost during a tuple transfer. We are currently researching mechanisms to reduce the atomicity guarantees of the transaction protocols so as to limit the window of vulnerability. To address data instability, we are defining new classes of data to reflect the state of the transfer where the connection was lost. To complement these data classifications, we are also describing policies for accessing the possibly inconsistent data which range from conservative approaches regarding availability to possible duplication. Furthermore, reconciliation protocols must be put in place to resolve inconsistencies when a connection is established again.

Another important consideration of the mobile environment is security. The current implementation of LIME does not specifically address security, however we recognize that security is a major concern, particularly in the ad hoc environment and have several ideas about how to integrate security into LIME. First, the notion of a private tuple space can already be exploited to keep data separate. Alternately, it should be possible to share data in a limited fashion, for example a tuple space (or at a more fine-grained level, the tuples themselves) can be augmented with with an access control list. The modular design of the implementation means that these changes should be localized to specific parts of the transaction protocol and to the `LimeTSMgr`. From the application perspective, many more parameters will be tunable, providing added flexibility.

6 Conclusions

LIME is our first attempt at designing middleware for mobile systems based on the theme of coordination. The notion of transiently shared tuples 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 redefined, 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 confines 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.

References

[1] LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conf. on Software Engineering*, pages 368–377, May 1999.

[2] G. Blair, N. Davies, A. Friday, and S. Wade. Quality of Service Support in a Mobile Environment: An Approach Based on Tuple Spaces . In *Proc. of the 5th IFIP Int. Wkshp. on Quality of Service (IWQoS '97)—Building QoS into Distributed Systems*, pages 37–48, May 1997.

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

[4] IBM. T Spaces Web page. `http://www.almaden.ibm.com/cs/TSpaces`, 2000.

[5] JavaSpaces. The JavaSpaces Specification web page. `http://www.sun.com/jini/specs/js-spec.html`, 1999.

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

[7] 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.

[8] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Operating Systems Review*, 29(5):172–183, 1995.

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