Using LIME to Support Replication for Availability in Mobile Ad Hoc Networks

Amy L. Murphy[1] and Gian Pietro Picco[2]

[1] Faculty of Informatics, University of Lugano, Switzerland
E-mail: amy.murphy@unisi.ch
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
E-mail: picco@elet.polimi.it

Abstract. Mobile ad hoc networks (MANETs) define a challenging computing scenario where access to resources is restrained by connectivity among hosts. Replication offers an opportunity to increase data availability beyond the span of transient connections. Unfortunately, standard replication techniques for wired environments mostly target improvements to fault-tolerance and access time, and in general are not well-suited to the dynamic environment defined by MANETs.
In this paper we explore replication for mobility in the context of a veneer for LIME, a Linda-based middleware for MANETs. This veneer puts into the hands of the application programmer control over what to replicate as well as a set of novel replication and consistency modes meaningful in mobile ad hoc networks. The entire replication veneer is built on top of the existing LIME model and implementation, confirming their versatility.

1 Introduction

Mobile ad hoc networks (MANETs) recently emerged as a technology enabling distributed computing in untethered scenarios. Typical applications exhibiting novel coordination patterns range from collaborative work in impromptu meetings to coordination of rescue teams in a disaster recovery setting. As MANETs are characterized by fluid topology and transient connectivity, they undermine the assumptions traditionally made by established distributed computing methods, algorithms, and technologies, and in many cases demand new solutions taking into account the *opportunistic* nature of communication in the mobile ad hoc environment.

In this paper we focus on the issue of data replication. In traditional distributed systems, replication is usually employed for fault tolerance or performance by exploiting, respectively, the redundancy and the locality of data copies. In a mobile environment, and especially in MANETs, replication achieves data availability by enabling access to the data beyond the span of a transient connection. Moreover, traditional replication schemes usually aim at providing a high degree of consistency in the way clients perceive access to replicas. Consistency

protocols introduce synchronization and therefore communication and computational overhead, often make assumptions about the placement of replicas, and usually assume stable connectivity—characteristics that often clash with the requirements of MANET applications.

The work we present here tackles replication from a different angle. To begin with, the data we replicate are tuples belonging to a distributed tuple space system. More specifically, we developed our replication strategy as a veneer on top of the federated tuple space provided by LIME [14, 18, 21], a middleware we developed expressly for the MANET environment. In LIME, each mobile host[3] carries a local tuple space; an agent running on a given host is given access to a global, federated tuple space constituted by the "fusion" of all the tuple spaces belonging to the hosts in range. Therefore, in a mobile setting the content of this global tuple space changes dynamically based on the current connectivity.

In LIME, a tuple exists at a single location, and becomes available only for the time span during which the host carrying it is transiently connected to the rest of the system. Following what has been done in related work targeted to improving fault tolerance or access performance, one could support replication by copying tuples across machines and providing transparent, consistent access to them (e.g., by properly serializing read and write operations). As we mentioned, however, the available techniques need substantial adaptation to become usable in the MANET environment. Replicating the whole tuple space is likely to generate too much overhead, and keeping the tuple space consistent is hard if not impossible in the presence of hosts that can disconnect arbitrarily and possibly never reconnect.

Instead, our focus here is a simple and yet effective mechanism to increase data availability. We achieve this by providing the programmer with the ability to specify *replication profiles*, denoting the tuples of interest for the application. The underlying replication system exploits these profiles to opportunistically and automatically create a local replica whenever a matching tuple is encountered in the system. Differently from traditional replication systems, where replication is entirely transparent to the programmer, in our model the programmer is aware of whether a tuple is the original copy or a replica, as in the uncertain environment we target this information is often key in determining the confidence to be placed in the data being communicated. We do, however, provide guarantees about when a tuple is updated to a newer version, and provide options for specifying constraints on how this update is performed (e.g., only from the master copy or from any replica). Notably, replicating from replicated data instead of only from original data enables transitive models of coordination where replicas epidemically spread in the system, even if the master copy is not available.

In the work we present here, all these aspects are folded into the LIME application programming interface (API). This provides the programmer with the ability to query for and react to conventional tuples as well as replicas—and to do this in a distributed fashion regardless of connectivity, by exploiting the LIME

[3] LIME actually provides support for *both* mobile agents and hosts, therefore encompassing both logical and physical mobility.

federated tuple space. Moreover, our replication veneer is entirely built on top of the original LIME middleware, exploiting in particular its reactive features, and therefore providing additional evidence of LIME's versatility and expressiveness.

The paper is structured as follows. Section 2 provides the reader with the necessary background, by surveying related work and concisely illustrating the features of the LIME middleware. Section 3 illustrates the motivations behind our approach by leveraging our previously reported experience [13] in developing context-aware applications with LIME. Section 4 presents the replication model we define, together with the API provided to the programmer. Section 5 reports about the design and implementation of our replication veneer. Section 6 elaborates on the previous sections and highlights opportunities for alternative designs. Section 7 contains brief concluding remarks.

2 Background

In this section we first survey related work concerning replication applied to tuple space systems or in the mobile environment, and then provide a concise overview of the LIME model and middleware.

2.1 Related Work

With the growing interest in MANETs, strategies have been investigated for hoarding data and keeping it accessible when hosts are disconnected from data servers. Coda [12] was among the first hoarding systems, using user profiles to decide what to hoard and requiring user intervention for conflict resolution. Bayou [22] and IceCube [11] maintain consistency by logging changes and ensuring log serializability. Similarly, [2] proposes a middleware service for increasing data availability among groups of users according to user-specified profiles. It applies a conservative coherence protocol to ensure data is accessed consistently among group members in the presence of data updates and allows any member to update data as long as the object's unique consistency token is available. Instead, the work presented here is based on less constraining assumptions about the connectivity among hosts and thus their ability to reconcile inconsistent data. This choice simplifies the model, yet still addresses the needs of a wide range of applications where data availability is necessary even during disconnection.

In fixed networks, some distributed implementations of Linda investigated replication for fault tolerance [1, 19, 23] and strategies have been proposed for maintaining consistency among the distributed data [5]. All of these approaches assume that disconnection of a host is a failure and require extensive network communication to maintain data consistency. Both of these assumptions are fundamentally challenged by the MANET environment in which disconnection is an expected event and wireless communication is more constrained than wired communication.

Two interesting alternatives are GSpace [20] and PeerSpaces [3]. GSpace proposes a system-level framework for managing the trade-offs between replication

for availability versus performance. PeerSpaces supports shared data spaces in the peer-to-peer environment and introduces replication for availability, but only for read-only data. Nonetheless, both systems target large scale networks, and are not applicable in MANET environments.

MobiSpaces [9] targets replication of tuple spaces among mobile devices and shares many goals with the work presented here. It allows replication from non-primary sources and accepts *interest profiles* from mobile users to control what data is replicated. However, The MobiSpaces assumes a single, master tuple space that serves as the primary holder of the data and determines the legal sequence of access to tuples based on a form of causal ordering. In contrast, our work assumes each mobile device can be the primary holder data and we do not assume any causal ordering semantics among access to tuples, keeping our semantics closer to the original Linda.

2.2 LIME: Linda in a Mobile Environment

The LIME model [18] defines a coordination layer for applications that exhibit logical and/or physical mobility, and has been embodied in a middleware [14] available as open source at `http://lime.sourceforge.net`. LIME borrows and adapts the communication model made popular by Linda [10].

In Linda, processes communicate through a shared *tuple space*, a multiset of tuples 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. Tuples are anonymous, thus their removal by **in**(p), or read by **rd**(p), takes place through pattern matching on the tuple content. 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,?float>` are formals. Formals act like "wild cards" and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, selection is non-deterministic.

Linda characteristics resonate with the mobile setting. Communication is implicit, and decoupled in *time* and *space*. This decoupling is of paramount importance in a mobile setting, where the parties involved in communication change dynamically due to migration, and hence the global context for operations is continuously redefined. LIME accomplishes the shift from a fixed context to a dynamically changing one 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.

Transiently shared tuple spaces. In LIME, a mobile unit accesses the global data context only through a so-called *interface tuple space* (ITS), permanently and exclusively attached to the unit itself. The ITS, accessed using Linda primitives, contains tuples that are physically co-located with the unit and defines the only data available to a lone unit. Nevertheless, this tuple space is also *transiently*

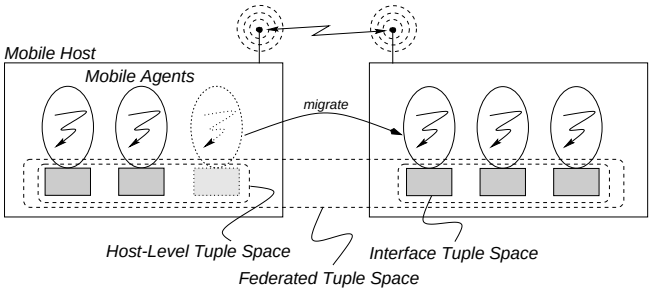


Fig. 1. Transiently shared tuple spaces encompass physical and logical mobility.

shared with the ITSs belonging to the mobile units currently accessible. Upon arrival of a new unit, the tuples in its ITS are logically 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. Similarly, the departure of a mobile unit results in the *disengagement* of the corresponding tuple space, whose tuples are no longer available through the ITS of the other units.

Transient sharing of the ITS is a very powerful abstraction, providing a mobile unit with the illusion of a local tuple space containing tuples coming from all the units currently accessible, without any need to know them explicitly. Moreover, the content perceived through this tuple space changes dynamically according to changes in the system configuration.

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 defined. Figure 1 shows how transient sharing may take place among mobile agents co-located on a given host, and among hosts in communication range. Mobile agents are the only active components, and the ones carrying a "concrete" tuple space; mobile hosts are just roaming containers providing connectivity and execution support for agents.

Operations on the transiently shared tuple space of LIME include those already mentioned for Linda, namely **out**, **rd**, and **in**, as well as the probing operations **rdp** and **inp** whose semantics is to return a matching tuple or return `null` if no matching tuple exists at the time the query is issued. For convenience LIME also provides the bulk operations **rdg** and **ing** that return a set of tuples that match the given pattern. If no matching tuples exist, the set is empty.

Restricting Operation Scope. The concept of transiently shared tuple space 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 specifically addressing configuration changes, but sometimes applications may need to address explicitly the distributed nature of data for performance or optimization reasons. For this reason, LIME extends Linda operations with scoping parame-

ters, expressed in terms of agent or host identifiers, that restrict operations to a given projection of the transiently shared tuple space. For instance, $\mathbf{rd}[\omega, \lambda](p)$ looks for tuples matching p that are currently located at ω but destined to λ. LIME allows ω to be either a host or an agent, enabling queries over the entire host-level tuple space or only over the subset pertaining to a specific agent.

Reacting to changes. In the dynamic environment defined by mobility, reacting to changes constitutes a large fraction of application design. Therefore, LIME extends the basic Linda tuple space with a notion of *reaction*. A reaction $\mathcal{R}(s, p)$ is defined by a code fragment s specifying the actions to be performed when a tuple matching the pattern p is found in the tuple space. A notion of *mode* is also provided to control the extent to which a reaction is allowed to execute. A reaction registered with mode ONCE is allowed to fire only one time, i.e., it becomes automatically deregistered after its execution. Instead, a reaction registered with mode ONCEPERTUPLE is allowed to fire an arbitrary number of times, but never twice for the same tuple. Details about the semantics of reactions can be found in [15]. Here, it is sufficient to note that two kinds of reactions are provided. *Strong reactions* couple in a single atomic step the detection of a tuple matching p and the execution of s. Instead, *weak reactions* decouple the two by allowing execution to take place eventually after detection. Strong reactions are useful to react locally to a host, while weak reactions are suitable for use across hosts, and hence on the entire transiently shared tuple space.

LIME provides a number of additional features, including the ability to output a tuple into the tuple space of a different agent with the $\mathbf{out}[\lambda](t)$ operation, and to obtain information about the host, agents, and tuple spaces currently present in the system through a specialized LimeSystem tuple space. However, as these and other features are not central to the work described here, we redirect the reader interested in a comprehensive description to [15], which also includes a formal semantics of the LIME model.

3 A Motivating Example

In this section we discuss the motivation behind our particular approach to improving data availability through replication by leveraging off our previously reported experience [13] with the LIME tuple space primitives to develop context-aware applications.

In [13] we discussed the design of TULING, a location-aware application supporting collaborative exploration of geographical areas, e.g., to coordinate the help in a disaster recovery scenario. Users are equipped with portable computing devices and a localization system (e.g., GPS), are freely mobile, and are transiently connected through ad hoc wireless links. The key functionality provided is the ability for a user to request the visualization of the current location and/or trajectory of any other user, provided wireless connectivity is available towards her. Additionally, applicative data (e.g., images or notes) can be annotated with location information before being stored in the tuple space, and

therefore searched based on context. The implementation exploits tuple spaces as repositories for context information—i.e., location data in this case. The LIME primitives are used to seamlessly perform queries not only on a local tuple space, but on all the spaces in range. For instance, a user's location can be determined by performing a **rdp** operation for the location tuple associated to the given user identifier. Similarly, reactions can be used to trigger some behavior when a user changes her location.

The thesis of the paper was simple and yet relevant: tuple spaces can be successfully exploited to store not only the application data needed for coordination, but also data representing the *physical context*. The advantage is the provision of a single, unified programming interface—the LIME coordination primitives—for accessing both forms of data, therefore simplifying the programmer's chore.

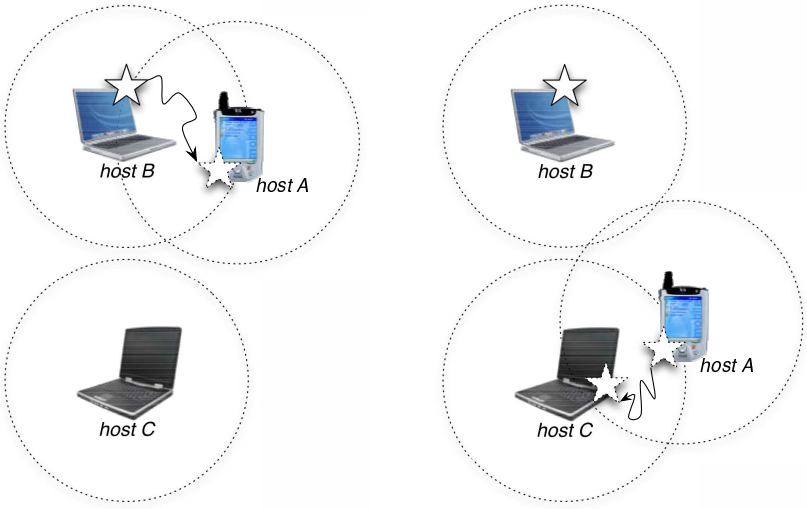
Nevertheless, the paper also elicited a number of shortcomings in the primitives and mechanisms traditionally provided by tuple space systems in general, and by LIME in particular. For instance, it evidenced how the matching by equality traditionally provided by Linda is not sufficiently expressive for context-aware applications. This observation provided the main motivation for a recent extension [17] to the LIGHTS tuple space engine [16] at the core of LIME.

Similarly, the motivation for the particular flavor of replication presented here can be found among the "lessons learned" we reported after developing TULING, as evident in the following excerpt (see [13], p. 276):

> Another feature to consider adding to LIME is replication. In TULING, the previous locations of the other components are effectively replicated at the application level, to enable their visualization. Location information, however, is not duplicated within the tuple space. Therefore, if A copies B's history, and then later meets C, the information about B is outside the tuple space and therefore not accessible to C. Several efforts in the mobile ad hoc community have looked at the issue of replication [...], but none of the solutions is immediately applicable to the tuple space environment.

This excerpt captures the essence of the problem. The reactive primitives provided by LIME can be used effectively to copy location information as soon as it becomes available through the federated space and carry on the associated behavior, but replication occurs *at the application level* and therefore does not enable further sharing of the information acquired. The desired scenario is instead the one shown in Figure 2.

Clearly, one could write a reaction reinserting the location tuple in the local tuple space, but care must be taken in "tagging" this copy so that it does not get reacted again, locally or remotely. Therefore, rather than simply build this behavior into TULING, we created an application-independent middleware layer on top of LIME to support this kind of replication. As we detail in the next section, the goal is to give users the ability to declare the patterns of tuples to be replicated, when and how to replicate, and whether and how the replica should be updated in the presence of new values.



(a) A and B engage: B's tuple is replicated in A's tuple space.

(b) A disengages from B and engages with C: the replica of B's tuple is copied in C's tuple space.

Fig. 2. A motivating scenario. Dashed outlines indicate replicated data.

Obviously, location is only one of the many kinds of data that is meaningful to replicate. Besides other contextual data (e.g., energy level, temperature, light, and so on), replication of application data is useful as well. For instance, TULING users can share images, e.g., pictures useful for a damage assessment of buildings in an earthquake scenario.

By using the features of our middleware, programmers can leverage replication in many ways. Not only can the programmer specify replication profiles such as "*replicate all pictures of buildings between the* 5^{th} *and* 7^{th} *avenue*", but she can also choose whether such pictures should be downloaded only from the user that took them or from anyone. The implications of what looks like a trivial choice are amplified by the integration of replication with the transiently shared tuple space abstraction provided by LIME. Indeed, in the first case our system implicitly supports a sort of "hoarding" [12] from information producers, which materializes the requested information in the tuple space of a potential consumer tuple space as soon as it becomes available in the system. In the second case, by allowing duplication from any replica, information can flow even in the absence of a connected path between its producer and its consumers, by "hopping" opportunistically from one machine to another whenever connectivity becomes available. This pattern of information dissemination, somewhat reminiscent of epidemic protocols [7], is more closely related to the disconnected transitive communication model explored in [4] and, more recently, by the networking community under the notion of *delay tolerant network* [8].

Therefore, besides enhancing data availability, our replication layer can be regarded (and exploited) as a building block for a different form of coordination that extends the transient communication enabled by LIME by removing the need for interacting parties to be present at the same time.

4 Replication Model and API

Throughout the development of the model and implementation, our goal was to put control over replication in the hands of the programmer while keeping the interface straightforward and easy to use. The primary issues to address are updating tuple contents to allow tuples to evolve over time, identifying what to replicate based on user input, and updating replicas when master tuples are updated and connectivity allows. In this section we address each of these issues as they relate to the model, concluding with a description of the API.

Updating tuples. Some applications logically create multiple versions of the same piece of information with each successive version invalidating the previous. For example, location data in TULING constantly changes as a host moves through space. Each new location represents an update, replacing the now-irrelevant previous location. In most tuple space systems, it is only possible to remove the old data and insert new data, losing any logical connection between the two. Instead, it is meaningful to allow the data to be *changed*, associating it with the old data and at the same time identifying that it has been updated.

This is precisely what we provide, allowing the user to specify a template for the old data together with the actual new data. The new data is distinguished from the old with a version number. This mechanism also serves as a building block upon which consistency between master and replicas is managed.

Identifying what to replicate. As our goal is to improve availability of data for users rather than to improve the performance of the system, our replication mechanism is driven by user input in the form of replication profiles. These are composed of the template specifying the tuples to be replicated, together with the replication and consistency modes controlling the replica creation and update.

As we discussed in Section 2.2, the scope of LIME operations can be restricted to the tuple space associated to a single host or agent, by properly setting the current and destination location of matching tuples. We provide a similar feature for replication, therefore enabling the programmer to to replicate matching data only if it belongs to the specified tuple space projection, e.g., tuples belonging to a given host. Alternately, this tuple location information can be left unspecified in the template, therefore enabling replication of tuples from any connected host. One notable exception, however, is that replication of local tuples is suppressed because it does not improve information availability.

Another dimension of the replication profiles is whether replicas should be made only from original data or also from replicas held by other users. Because no

single policy holds for all applications, we place this decision in the hands of the programmer in the form of a *replication mode* as shown in Figure 3. Replication mode MASTER makes replicas *only* from the original data while mode ANY allows a replica to be made indifferently from the original or a replica.

To make this more concrete, consider an extension of the case outlined in Section 3 in which host A replicates B's location information and then later meets C. Without replication inside the tuple space information about B is not available to C, as in TULING. However, with the mechanism described here and replication mode ANY, this information is available as replica tuples carried inside A's tuple space. This enables transitive communication of data as shown in Figure 2(b) even though B and C have never been in communication range.

Updating replicas. When a tuple is updated, a new version is created that makes any replicas of this tuple out of date. If connectivity exists between the holder of the master tuple and the holder of the replica it is possible to update the replica to the new version, however this comes at the cost of the data transfer. Depending on the type of data, it is reasonable to keep the replica out of date or to update it. Location information, for example, is typically small and most useful if kept up to date. Large documents, instead, may remain useful even if they are slightly out of date. Therefore, our model allows the user to specify the policy for keeping replicated data consistent with the original. As shown in Figure 3, three possibilities are provided: NEVER, which never updates a replica, MASTER, which updates only from the master version of the tuple, and ANY, which updates from master or replica versions. If a replica is updated, its previous copy is deleted from the system.

To continue with the location example, by using replication mode ANY and consistency mode ANY at host A, when it engages with C as in Figure 2(b), C will have access to the most recently known location of B from when A and B were last connected. Although this is likely to be out of date with respect to the actual, current location of B, is it the best that can be done in the mobile environment with transient connections.

Replication API. For the programmer familiar with LIME, using replication requires minor changes to deal with extensions of tuple and template formats and new operations for dealing with replication profiles. The primary access to the tuple space is through an instance of the `ReplicableLimeTupleSpace`, as shown in Figure 4. The operations here retain the same meaning as in the

Replication Mode	MASTER	The first replica must be made from the master
	ANY	The first replica can be made from any tuple
Consistency Mode	NEVER	Replicas must never be updated
	MASTER	Replicas must only be updated from their master
	ANY	Replicas can be updated from any newer version

Fig. 3. Replication and consistency modes.

```
public class ReplicableLimeTupleSpace {
  public ReplicableLimeTupleSpace(String name);
  public boolean           setShared(boolean isShared);
  public ReplicableTuple   out(ITuple t);
  public ReplicableTuple   out(AgentLocation destination, ITuple t);
  public ReplicableTuple   in(ReplicableTemplate p);
  public ReplicableTuple   inp(ReplicableTemplate p);
  public ReplicableTuple[] ing(ReplicableTemplate p);
  public ReplicableTuple   rd(ReplicableTemplate p);
  public ReplicableTuple   rdp(ReplicableTemplate p);
  public ReplicableTuple[] rdg(ReplicableTemplate p);
  public ReplicableRegisteredReaction[] addStrongReaction(ReplicableLocalizedReaction[] rlr);
  public ReplicableRegisteredReaction[] addWeakReaction(ReplicableReaction[] rr);
  public void removeStrongReaction(ReplicableRegisteredReaction[] rrr);
  public void removeWeakReaction(ReplicableRegisteredReaction[] rrr);
  // REPLICATION-SPECIFIC OPERATIONS
  public ReplicableTuple          change(ReplicableTemplate p, ITuple t);
  public RegisteredReplicaRequest addReplicaRequest(LimeTemplate p,
                                                    int replicationMode,
                                                    int consistencyMode);
  public void                     removeReplicaRequest(RegisteredReplicaRequest r);
}
```

Fig. 4. The class `ReplicableLimeTupleSpace`.

original `LimeTupleSpace`, however the parameters are changed to deal with the additional information maintained for replication. Specifically, tuples, templates, and reactions have been replaced with their "replicable" counterparts, as seen in the figure.

For space reasons, we do not show the interfaces of all public classes here, however it is worth noting that the `ReplicableTemplate` allows the user to specify whether the tuple returned should be a MASTER, REPLICA, or ANY. Furthermore, `ReplicableTuple` exposes the `isMaster` method to allow the user to distinguish if the tuple returned from the tuple space is a master or not.

Reactions are also extended with respect to what available in LIME. Namely, when specifying a `ReplicableReaction` and a `ReplicableLocalizedReaction`, the user must identify the reaction mode. In addition to the ONCE mode which reacts one time before deregistering, we provide ONCEPERREPLICA and ONCEPERCHANGE. The former allows the user to react one time for each tuple, but not for each version of that tuple. The latter reacts also to each version.

The `ReplicableLimeTupleSpace` offers three new methods not present in the `LimeTupleSpace` to support tuple updating and replication. The `change` method accepts as parameters the template of the tuple to change and the contents of the new tuple. If no tuple matches the template, no change is made to the tuple space. Similar to the `out` operation, `change` operates only on the *local* tuples contained in the tuple space of the agent issuing the operation. Moreover, only master tuples can be selected and changed. The last two methods, `addReplicaRequest` and `removeReplicaRequest` allow the user to activate replication for the specified replication profile (template, replication mode, and consistency mode), and stop it, respectively.

5 Design and Implementation

Implementing the replication model just described was accomplished as a combination of two application-level packages above LIME, requiring no changes to LIME itself. The two layers support tuple versioning and tuple replication, respectively. Internally, each layer is implemented as a wrapper around the lower layer. Specifically, the `VersionedLimeTupleSpace` wraps an instance of `LimeTupleSpace` and the `ReplicableLimeTupleSpace` of Figure 4 wraps a `VersionedLimeTupleSpace`. Each layer implements the operations visible to the user by adapting and delegating them to the layer below. It should be noted that in a federated system, all tuple spaces must be of the same type, e.g., it is not possible to federate a `LimeTupleSpace` with a `VersionedLimeTupleSpace`.

In this section we describe the key components of each layer, focusing primarily on the replication layer.

5.1 Versioning

The primary functionality of the versioning layer is to support the **change** operation, allowing tuples to be updated instead of replaced. This requires changes both to the tuple format and to the primary tuple space operations.

Tuple format. Versioning of tuples requires that the new version both be associated with the old and distinguished as newer. The former is accomplished by assigning a tuple identifier to each newly created tuple. This identifier is simply prepended to the user data before the tuple is passed to LIME. When a tuple is updated, the new version uses the same tuple identifier. To identify the relative newness of a tuple, we also insert a version number, incremented each time the tuple is changed. To clarify, if the user requests insertion of the tuple $\langle data \rangle$, the versioning layer creates the following and passes it to LIME:

$$\langle data \rangle \rightarrow \langle tupleID, versNum, data \rangle$$

Operations. To support updating tuples, the API is extended with the `change` operation, similar to that shown in Figure 4, accepting the template and new data as parameters. Internally, `change` is implemented by performing an `inp` on the embedded `LimeTupleSpace` to remove a tuple matching the pattern. If a tuple is returned, the versioning layer extracts the tuple identifier and version number, increments the version number, prepends both to the tuple, and issues an `out` to insert the new tuple.

Similar to `ReplicableLimeTupleSpace`, the `VersionedLimeTupleSpace` operations have been modified to accept "versioned" tuples and templates to address the tuple identifier and version number. Reactions have also been modified to use two new modes in addition to ONCE, namely ONCEPERID and ONCEPERVERSION to react only one time per tuple identifier, or one time to each version of each tuple. These reactions, with their enhanced modes, are actually the main building block for implementing replication, as described next.

5.2 Replication

Building replication on top of the versioning layer involved much the same process as implementing the versioning layer on top of LIME. It requires fields to be added to tuples and new operations to accept and implement replication profiles. The new operations, however, have already been described in Section 4: here, we describe their implementation.

Tuple format. The first design decision we faced was whether to keep the replica tuples in the same tuple space as the master tuples or to divide the two explicitly. We chose the former, opting to tag each tuple with a new field (*isReplica*) to distinguish whether it is a master or a replica. A nice side effect of this choice is that a query for a tuple without explicitly specifying replica or master requires only one operation, with the aforementioned field set to formal. Dividing the tuples would require issuing operations on both spaces.

In addition to the *isReplica* field, tuples are also extended with two fields representing the current and destination locations of the master tuple. To understand the need for these new fields, it is important to remember that LIME uses current and destination fields to identify the *current* location of a tuple and whether it should be migrated into the tuple space of a different, remote agent upon engagement. Because replica tuples are normal LIME tuples, location information is also maintained for all replica tuples internally to LIME. However, this information reflects the current and destination of the replica tuples, namely the agent that requested the replication, not the original location of the master tuple. Because the user may need to know this original location, we provide accessor methods on `ReplicableTuple` and append fields to all user tuples to represent the original current and destination locations of the master tuple as in the following:

$$\langle data \rangle \rightarrow \langle origCur, origDest, isReplica, data \rangle$$

Note how the tuple we obtain is then passed as a "data" tuple to the versioning layer where it gets extended with other fields, as described in Section 5.1.

Implementing replication. The core of the implementation is the internal mechanism used for replication. Our model requires creation and updating of replicas. Both operations occur in reaction to the appearance, in the federated tuple space, of new tuples matching the specified pattern and conforming to the replication and consistency modes. Therefore, implementing reaction with a set of (versioning-layer) reactions over master and replica tuples is a natural approach.

Implementing a replication request for a given profile involves installing a reaction watching for master tuples and, depending on the replication and consistency modes, possibly one for replica tuples as well. When the reaction fires with a new tuple, the listener for that reaction must take the appropriate action to keep the replicas inside the tuple space in line with the replication profile.

Consider, for instance, a case where the programmer requests a replication profile with a MASTER replication mode and a NEVER consistency mode. In this

Replication Mode	Consistency Mode	Master listener	Replica listener
MASTER	NEVER	*Keep*	—
MASTER	MASTER	Keep or update	—
MASTER	ANY	Keep or update	Discard if first, else update
ANY	NEVER	*Keep*	*Keep*
ANY	MASTER	Keep or update	Keep
ANY	ANY	Keep or update	Keep or update

Fig. 5. Implementing replication using reactions belonging to the versioning layer. For all combinations of replication and consistency mode, the listeners describe the actions performed when the reaction fires with a tuple on either a master or replica tuple. Italicized listeners are mode ONCEPERID, all others are ONCEPERVERSION.

case, only the reaction watching for master tuples is needed, as subsequent versions are uninteresting. For the same reason, the corresponding listener needs to react each time a new master tuple is inserted, but not when it is updated. Therefore, the reaction (with the semantics defined in Section 5.1 for the versioning layer), must be installed with a ONCEPERID reaction mode. The reaction takes care of sending matching tuples from their owner to the requesting host when connectivity is available. However, LIME ensures also, through its engagement protocol, that the reaction is installed when hosts initially come into contact and that it remains enabled while the hosts are connected. Therefore, because LIME deals with the distribution and installation of listeners as connectivity changes, the replication layer can simply use this functionality, significantly reducing its own complexity.

To round out the other actions that must be taken to effect the various replication profiles, Figure 5 shows all combinations of the replication and consistency modes and the actions of all listeners for master and replica tuples. The other listeners for master tuples differ from the one we just described because they must also update the replica if it already exists. Therefore, the listener "Keep or update" updates the replica tuple if it already exists, or creates one if not. The combination MASTER/ANY utilizes a third type of listener that does not allow creation of the first replica from another one, but instead uses replicas only to update existing ones. Finally, all the combinations requiring updates upon a change in version number utilize ONCEPERVERSION reactions in order to capture all updates.

6 Discussion

Our current implementation, albeit fully working according to the design just described, must be considered as a proof-of-concept prototype demonstrating the feasibility of our ideas. A number of improvements and additional features can be introduced, some of which are discussed in the following.

Communication overhead. The current implementation just described may generate unnecessary overhead when the network is dense and stable and many nodes requested the replication mode ANY. Consider a host joining the system with replication mode ANY. By virtue of engagement its replication reactions are propagated to the other hosts, where they are immediately evaluated and return any matching replicas the hosts possess. However, if in turn the replication mode on the other hosts is ANY as well, the same process unfolds in the opposite direction. This causes not only new replicas brought by the joining host to be communicated to the other hosts, but also the newly inserted replica to follow the same destiny. This last replica is most likely discarded, because it carries information already present in the network. In the scenarios mentioned above, this unnecessary extra step my cause a significant contribution to overhead.

A couple of points are worth making, however. First of all, in scenarios where the system contains many hosts enjoying rather stable connectivity, the replication mode ANY is bound to generate a lot of traffic anyway, since everybody is likely to be up-to-date with respect to the system. Consider an impromptu meeting: a replication mode MASTER is probably the best choice, allowing each meeting attendees to obtain a copy of the document as soon as the latter is published by its owner. Instead, the mode ANY is well-suited to address the sparse scenarios typical of many MANET applications, where very few nodes are connected at any given time but over time overall system connectivity is provided as a consequence of movement and opportunistic interaction. For instance, in a disaster recovery scenario the members of the exploration team may be connected only transiently and unpredictably, and yet be able to get the images and notes posted by fellow members without ever being connected with them, thanks to transitive replication of replicas of the original documents. As we pointed out in Section 3, these scenarios are similar to those targeted, at the network layer, by disconnected transitive communication [4] and delay tolerant networks [8].

From an implementation standpoint, it is worth saying that there are ways to remove the aforementioned extra communication. A quick-and-dirty solution is simply to timestamp replicas upon their insertion in the tuple space, and use this time information to defer its propagation by a time T, under the assumption that the rest of the system (from where it came in the first place) is already aware of it. The question is clearly how to set the deferring time T: values too small reduce the benefit of the optimization if the network is stable, while in dynamic scenarios values too high may prevent propagation of the replica to hosts that recently joined the system. More sophisticated mechanisms (e.g., piggybacking lists of reacted-upon replicas) can be implemented, but at the cost of building replication management directly into the LIME system. Indeed, our mechanism is based on LIME's ONCEPERTUPLE reactions, which are not aware of replication and simply react to the presence of a new tuple. In our current prototype we aimed instead at preserving the independence of the replication layer from the base middleware, in an effort to foster separation of concerns and modularity.

Ultimately, the need for more optimized communication must be weighed against the deployment scenario and the way applications use replication. We

contend that the design and implementation we chose is appropriate for the assumptions made by the deployment scenario and application examples motivating this work, as well as for the "exploratory" goals of the research we report. Its exploitation in scenarios characterized by different assumptions may obviously require significant adaptation.

Automatic purging of local replicas. In the current implementation, replicated tuples are automatically inserted in the local tuple space, where they remain until the programmer expressly decides to remove them. This policy constitutes the most basic solution and meets the needs of simple applications. Nevertheless, in the presence of several replicas that need different treatment, their management may place considerable bookkeeping burden on the programmer.

Automatic purging can be easily defined by modifying `addReplicaRequest` to accept, in addition to replication and consistency mode, a "purge mode". For instance, the purge mode can be one of the values MANUAL, NUMBER, TIME. MANUAL corresponds to the current strategy, while the other two modes allow purging of the tuple space based on an additional parameter, i.e., either the maximum number of replicas for the specified template or their maximum permanence time in the tuple space. The implementation of this additional functionality is straightforward, and consists of either modifying the replication listeners to keep track of a counter (NUMBER) or associating a timer to the replica (TIME).

Removal of the master copy. As we discussed in Section 3, our motivation for tackling replication was provided by applications where the data being replicated is continuously updated (e.g., location). As such, we did not include mechanisms for dealing with the removal of a master tuple, and accordingly remove the replicas in the system. Moreover, in a true MANET scenario no assumption can be made about the movement of hosts, which therefore can remain out for range after the tuple withdrawal has been performed, complicating—or completely preventing—the reconciliation of the distributed tuple space.

One way to achieve this functionality is through the notion of a "death certificate" [6] associated to the tuple. A simple implementation of this notion is to update the master tuple by nullifying the application data, while retaining the version identifier. Hosts becoming connected with the master's host would get a copy of the master tuple, but its nullified content would signal that the master has actually been withdrawn from the tuple space, and therefore the replica must be withdrawn as well.

In a MANET environment, however, there is no guarantee that all the hosts owning a replica eventually become again part of the system, thus receiving the death certificate. Interestingly, a replication mode ANY somewhat helps in this respect, as its ability to epidemically spread information may bypass disconnections. At the same time, however, it complicates matters since the master host has absolutely no control over the replication of its tuple, which can be duplicated from a different replica. Therefore, the master faces the option of either keeping the death certificate *ad infinitum*, or removing it after a given time but potentially leaving some hosts with an inconsistent view of the tuple space.

Future work will investigate more sophisticated schemes able to strike better tradeoffs by making different assumptions about the movement of hosts.

7 Conclusions

Replication is a well-studied topic in distributed system, often applied also in the case of coordination languages exploiting tuple spaces. Nevertheless, the motivations for exploiting replication are typically to improve fault-tolerance or access time to tuples, while preserving a consistent view of the tuple space.

In this work, we took a different angle motivated by the desire to exploit coordination in the highly dynamic and disconnected environment characterizing MANETs, where the preeminent reason for replication is to ensure availability of the replicated data in the face of disconnection. Consistency is less important, as it may be difficult if not impossible to provide if no assumption about the movement of hosts is made. This particular flavor of replication may also be effectively exploited as a new form of coordination, based on interactions that occur without the coordination parties ever being connected at the same time.

We made these observations concrete by describing how they can be incorporated in LIME, an existing coordination middleware for MANETs. We defined an appropriate replication model, extended the LIME API with replication primitives, and built replication mechanisms as a veneer on top of LIME.

Acknowledgments. The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, by the National Research Council (CNR) under the IS-MANET project, and by the European Community under the IST-004536 RUNES project. The authors wish to thank Francesco Merlo and Massimo Montani for their implementation of the work described here.

References

1. D.E. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 1994.
2. M. Boulkenafed and V. Issarny. A middleware service for mobile ad hoc data sharing, enhancing data availability. In *Proc. of the Int. Middleware Conf.*, 2003.
3. N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. PeerSpaces: Data-driven coordination in peer-to-peer networks. In *Proc. of ACM Symposium on Applied Computing (SAC)*. ACM Press, 2003.
4. X. Chen and A.L. Murphy. Enabling disconnected transitive communication in mobile ad hoc networks. In *Proc. of the Workshop on Principles of Mobile Computing (POMC)*, pages 21–27, Newport (RI, USA), August 2001.
5. A. Corradi, L. Leonardi, and F. Zambonelli. Strategies and protocols for highly parallel Linda servers. *Software: Practice and Experience*, 28(14):1493–1517, 1998.
6. A. Demers et al. Epidemic algorithms for replicated data management. In *Proc. of the 6^{th} Symp. on Principles of Distributed Computing*, pages 1–12, 1987.

7. P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE Computer*, 37(5):60–67, May 2004.
8. K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. of ACM SIGCOMM*, pages 27–34. ACM Press, 2003.
9. A. Fongen and S. J. E Taylor. MobiSpace: A Distributed Tuplespace for J2ME Environments. In *14^{th} IASTED Int. Conf. on Parallel and Distributed Computing and Systems*, Arizona, USA, 2005.
10. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
11. A.-M. Kermarrec, A. Rowstron, M.Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *20^{th} Symp. on Principles of Distributed Computing (PODC)*, August 2001.
12. J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Systems*, 10(1):3–25, 1992.
13. A.L. Murphy and G.P. Picco. Using coordination middleware for location-aware computing: A LIME case study. In *Proc. of the 6^{th} Int. Conf. on Coordination Models and Languages*, LNCS 2949, pages 263–278. Springer, February 2004.
14. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proc. of the 21^{st} Int. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.
15. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A coordination middleware supporting mobility of hosts and agents. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 2006. To appear. Available at `www.elet.polimi.it/upload/picco`.
16. G.P. Picco. LIGHTS Web page. `lights.sourceforge.net`.
17. G.P. Picco, D. Balzarotti, and P. Costa. LIGHTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. In *Proc. of the 20^{th} ACM Symposium on Applied Computing (SAC05)—Special Track on Coordination Models, Languages and Applications*, pages 1134–1140, March 2005. Extended version to appear in the *Int. J. on Web Intelligence and Agent Systems (WAIS)*.
18. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21^{st} Int. Conf. on Software Engineering*, pages 368–377, May 1999.
19. J. Pinakis. *Using Linda as the Basis of an Operating System Microkernel*. PhD thesis, University of Western Australia, Australia, August 1993.
20. G. Russello, M. Chaudron, and M. van Steen. Dynamically adapting tuple replication for managing availability in a shared data space. In *Proc. of the 7^{th} Int. Conf. on Coordination Models and Languages*, LNCS 3454. Springer, April 2005.
21. Lime Team. LIME Web page. `lime.sourceforge.net`.
22. 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.
23. A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Digest of Papers of the 19^{th} Int. Symp. on Fault-Tolerant Computing*, pages 199–206, June 1989.