Exploiting Transiently Shared Tuple Spaces for Location Transparent Code Mobility

Gian Pietro Picco and Marco L. Buschini

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32, I-20133 Milano, Italy
Phone: +39-02-23993519, Fax: +39-02-23993411
E-mail: picco@elet.polimi.it,buschini@computer.org

Abstract. Code mobility greatly improves the flexibility of the architecture of a distributed application. However, currently available platforms do not exploit fully the potential of mobile code. For instance, remote dynamic linking of code is often restrained to a well-known site, and applications are prevented from manipulating their own code base.
In this paper, we use the notion of transiently shared tuple space, originally introduced in the LIME coordination model, to overcome these limitations. We allow tuples to contain classes, and tuple spaces become the code base associated to the loading mechanism in the mobile code runtime support. Transient sharing allows for location transparent retrieval of classes, and accommodates changes determined by the reconfiguration of the system, e.g., due to mobility. In doing this, we effectively define a new coordination approach that deals uniformly with code and data.
The presentation is completed by a proof-of-concept prototype, built by extending an existing Java-based mobile code toolkit.

1 Introduction

Code mobility [7] is increasingly considered among the options available to the designer of distributed applications. Mobile code enables run-time component relocation, hence decoupling a component from the location where it is executed. This separation provides several advantages, notably the potential for a better use of communication resources and the enhanced flexibility of the overall system.

Technologies supporting mobile code differ in the mechanisms provided and in the relocation styles supported[1]. However, code mobility is typically exploited to download code from a well-known source, that acts as a code repository, thus providing the ability to extend at run-time a remote executing unit. This solution has proven its validity in several domains, among which the downloading of Web applets from a server to a browser is probably the most popular and best known.

Nevertheless, this solution can be overly limiting in scenarios that are less stable and predictable, like those defined by mobility, and where it is not possible (or not practical) to expect to know statically the location where a code fragment

[1] For a survey of technologies, architectures, and applications of code mobility see [7].

will be found at run-time. Moreover, most of the middleware exploiting mobile code keeps it behind the scenes. Code relocation happens only as an indirect effect of some system event, like the need to resolve a class that is not found locally. Applications are almost never enabled to manage directly their code base, e.g., by explicitly adding or removing code fragments. Nevertheless, this latter capability is often useful to define code caching schemes.

In this paper, we explore an approach to deal with mobile code that: *i)* does not depend on the presence of one or more code repositories dispersed in the network, rather it enables applications to localize and retrieve code with various degrees of location transparency; *ii)* allows applications to manage explicitly the code fragments that are being relocated within the system.

To achieve this goal, we adopt a coordination perspective rooted in the idea of transiently shared tuple space introduced by LIME [15, 13], a middleware that adapts Linda to the domain of physical and logical mobility. In LIME, Linda's notion of a global and persistent tuple space is replaced by a transiently shared tuple space that contains the union of the tuple spaces belonging to the mobile units currently in range, and whose content is dynamically rearranged according to mobility. In this work, we explore the opportunities that arise when the LIME notion of transiently shared tuple space is coupled with mobile code mechanisms. The result is a coordination infrastructure that treats code and data in a uniform way, and enables the coordination of agents not only through the exchange of information, but also by direct manipulation of the agent behavior.

To prove the feasibility of our ideas we implemented a proof-of-concept prototype that extends the mobile code toolkit μCODE [14] with the ability to dynamically load classes in a location transparent way from a transiently shared tuple space. Nevertheless, it is not our intent to present here a final, full-fledged solution or system. Instead, by eliciting the synergies between coordination models and mobile code, and by showing an implementation path towards their realization, our ultimate goal is to spur further research about this topic.

The paper is structured as follows. Section 2 discusses the motivations of the work we present. Section 3 gives a brief overview of the LIME model. Section 4 discusses the advantages and opportunities disclosed by coupling a transiently shared tuple space with mobile code. Section 5 reports about the design and implementation of our prototype. Section 6 places our paper in the context of related work. Finally, Section 7 discusses ongoing and future work on the topic of this paper, together with some brief concluding remarks.

2 Motivation

Currently available support for mobile code is mostly limited to variations of the well-known class loading mechanism provided by Java[2]. In Java, the class loader is programmable, and allows for redefinition of most of the logic determining

[2] Since Java-based mobile code systems are by far the most common, and the systems considered in this paper are Java-based, we often use *class* instead of *code fragment*, although a lot of what follows is applicable also to systems that do not rely on Java.

how a class is retrieved at name resolution time. Systems exploiting mobile code typically specialize the Java class loader by defining alternative ways to retrieve classes, e.g., by downloading them from a specified site. This code on demand [7] approach is the one originally used to support Java applets in Web browsers and is increasingly being exploited also in middleware, e.g., for dynamic downloading of stubs in Java/RMI and Jini. Moreover, it is being exploited in mobile agent platforms, to allow an agent to travel with only a subset of the classes needed, and dynamically download the others on demand.

Unfortunately, in its most common incarnations this approach has at least two relevant drawbacks. First of all, the local *code base*, i.e., the set of classes locally available, is usually accessible only to the run-time support, and hence it remains hidden from the applications. Moreover, remote dynamic linking of code is usually limited to a well-known site acting as a centralized remote code base. In the following, we analyze these two limitations in more detail.

Lack of Dynamic Access to the Code Base. Usually, applications can intervene on the code base only statically, e.g., by specifying the value of the `CLASSPATH` environment variable. As a consequence, an application cannot query or modify at run-time the set of classes available locally or at a remote site. Nevertheless, this capability turns out to be useful in several situations. For instance, it would open up the ability for dynamic reconfiguration. The current code base could be inspected by a system administrator operating remotely, or even by a mobile agent co-located with the application, and upgraded with code fragments containing new functionality, or replacing obsolete ones.

Moreover, it would enable applications to define code caching schemes. Some amount of caching is already provided in Java-based systems, where the redefined class loader typically maintains a class cache to avoid unnecessary retrieval and definition of class objects already loaded. Nevertheless, the cache is typically filled only at name resolution time, after the bytecode retrieval and class object definition. Again, there is no way to explicitly and dynamically modify the cache content, e.g., by pre-loading classes that are known to be needed in the future.

As an example, imagine a monitoring agent installed on a remote host, e.g., to perform network management. A reasonable design for these systems is to place in the agent the core functionality for reporting data back to a centralized management station and to perform local recovery from some common anomalous situations, and let the agent retrieve on demand the code needed for handling exceptional situations. However, what if the management station determines that network is about to become partitioned, e.g., due to congestion? Access to the agent's code base would allow the management station to upload additional routines on the agent, to cope with the anomalous situation during the period of disconnection, before the latter actually occurs. Similarly, class caching could be beneficial in situations where only weak connectivity is available. For instance, a server communicating with a PDA over an increasingly noisy wireless link may decide to proactively push some of the application classes into the PDA's class cache, instead of letting the PDA download them when needed, and hence possibly during a likely disconnection.

Location-based Dynamic Linking. The other relevant drawback of current approaches to mobile code lies in the pairwise nature of the remote dynamic linking process. As we mentioned, missing classes are usually downloaded from a well-known site. For Web browsers, the site is the Web server from which the page was downloaded. For Java/RMI, it is the Web server whose URL is specified as a code base in the stubs associated to either the source or the target object of a remote method invocation. For mobile agent platforms, it is typically either the source of migration or a centralized code repository like in Aglets [11], Mole [16], or JumpingBeans [6], to cite some among the best known Java-based systems.

Granted, this simple mechanism already allows for unprecented levels of flexibility in deploying the code of a distributed application. Still, it appears to be a limitation in several situations, e.g., when the code repository is not available, when its location cannot be determined in advance, and generally in applications characterized by a high degree of dynamic reconfiguration.

An example is provided by the field of code mobility itself. One of the advantages often claimed in the literature for mobile code is the ability to support disconnected operations [7, 9]. In situations where communication with another machine over a network link should be minimized, e.g., because the communication link is noisy, subject to disconnections, expensive, or insecure, mobile code can be exploited to allow the sender to ship application code to the target machine, where it can perform processing on behalf of the sender during disconnection. This idea is brought to an extreme by mobile agents, which allow a whole executing unit (e.g., a thread) to roam autonomously in the network without the need for connectivity towards its sender. Clearly, the aforementioned schemes for dynamic downloading, that are indeed surprisingly a very common choice in mobile code and mobile agent platforms, actually hamper the use of these systems for supporting disconnected operation.

Another setting where location transparent class loading is likely to be key is provided by mobile computing and in particular by mobile ad hoc networks (MANET) [12], which bring network reconfiguration to an extreme by assuming that the fixed infrastructure is totally absent. The fluidity of the MANET environment is such that network functions like routing must be provided by the mobile hosts themselves. Similarly, the application layers usually favor a peer-to-peer architecture over a client-server one, since it is usually difficult to pick a stable spot to place a server. Essentially, the mobile hosts can count only on those resources that are present in the system at a given moment—and code is no exception. As a possible application of location transparent code mobility, imagine a scenario in the automotive application domain, where cars on a highway are part of a MANET. Mobile code could be exploited to propagate in an epidemic way upgrades to car maintenance or monitoring routines, while cars are in range, or even while they are passing in opposite directions.

Nevertheless, even in scenarios where connectivity is permanent and a code repository is always available, alternatives are often preferrable. An application executing at a given site and needing a class to proceed with execution, may actually find it on a site close by or even on the same site, simply because other

application components already downloaded it. Similarly, a mobile agent may find out that a needed class is present at a site nearby and either fetch it, or move to that site and link it locally, if fetching is somehow prevented. Linking from a statically determined location is simply too rigid a scheme.

In this work, we provide a way to expose the code base of the executing units belonging to a distributed application, and to couple it with a location transparent class loading mechanism supporting the linking of mobile code. We achieve this goal by building upon an existing coordination model and system called LIME, towards which we now turn our attention before describing in detail our approach in the remainder of the paper.

3 LIME: Linda in a Mobile Environment

The LIME model [15, 13] defines a coordination layer for applications that exhibit logical and/or physical mobility, and has been embodied in a middleware available as open source at `http://lime.sourceforge.net`. LIME borrows and adapts the communication model made popular by Linda [8].

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 $\mathsf{out}(t)$ operation. Tuples are anonymous, thus their removal by $\mathsf{in}(p)$, or read by $\mathsf{rd}(p)$, 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. A mobile unit accesses the global 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 context available to a lone unit. Nevertheless, this tuple space is also *transiently shared* with the ITSs belonging to the mobile units currently accessible. Upon arrival of a

new unit, the tuples in its ITS are merged with those, already shared, belonging to the other mobile units, and the result is made accessible through the ITS of each of the units. This sequence of operations, called *engagement*, is performed as a single atomic operation. 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.

Controlling Context Awareness. The idea 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 location parameters, expressed in terms of agent or host identifiers, that restrict the scope of operations to a given projection of the transiently shared tuple space.

The $\mathsf{out}[\lambda](t)$ operation extends out by allowing the programmer to specify that the tuple t must be placed within the tuple space of agent λ. This way, the default policy of keeping the tuple in the caller's context until it is withdrawn can be overridden, and more elaborate schemes for transient communication can be developed. The semantics of $\mathsf{out}[\lambda](t)$ involve two steps. First, the tuple t is inserted in the ITS of the agent calling the operation, like in a normal out. If the agent λ is currently connected, t is atomically moved into λ's ITS. Otherwise, the

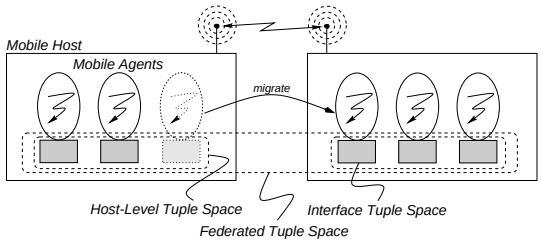

Fig. 1. Transiently shared tuple spaces encompass physical and logical mobility.

"misplaced" tuple t remains within the caller's ITS unless λ becomes connected. In this case, t migrates to λ's ITS as part of the engagement process.

Location parameters are also used to annotate the other operations to allow access to a slice of the current context. For instance, $\mathsf{rd}[\omega, \lambda](p)$ looks for tuples matching p that are currently located at ω but destined to λ.

Reacting to Changes in Context. In the dynamic environment defined by mobility, reacting to changes is a big 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. Details about the semantics of reactions can be found in [15, 13]. Here, it suffices 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 federated tuple space.

4 Transiently Shared Tuple Spaces as Code Bases

Coordination through LIME allows agents to access the global data space provided by the coordinated agents with varying degrees of location transparency, and yet to ignore the details of the system configuration. Nevertheless, similarly to other tuple space approaches, LIME tuple spaces are exploited to contain data, which are typically used either to provide the information necessary for coordination, or to store directly the information of interest for the application.

Instead, in this work we explore a very simple twist to the notion of tuple space. Essentially, we ask ourselves: What can we accomplish by allowing a LIME tuple space to contain classes, and by exploiting it as the code base associated to a class loading mechanism?

While the idea is very simple, its implications are far reaching. To begin with, one of the problems we highlighted in Section 2, namely, the need for abstractions that expose the code base to the application code, finds a natural solution. An agent can now manipulate its own code base through the primitives defined on LIME tuple spaces, since tuples containing classes can be treated just as ordinary data tuples, e.g., allowing class retrieval through pattern matching. Moreover, since a LIME tuple space is permanently and exclusively associated with its agent, when the latter moves its code base migrates along with it. Hence, tuple spaces provide a natural way to represent the code and state associated to a mobile agent, and to deal with the relocation of both in a uniform way.

However, transiently shared tuple spaces push the advantages one step further. Since, as we described, a LIME agent can share its tuple spaces with those belonging to other agents in range, each agent will have access to a code base that is potentially much bigger than its own. Transient sharing effectively stretches the boundaries of an agent code base to an extent possibly covering the whole

system at hand. Hence, the agent code base effectively becomes distributed, and its content dynamically and automatically reconfigured according to host connectivity and agent migration, according to the semantics of LIME.

This use of transiently shared tuple spaces solves the other problem mentioned in Section 2. A proper redefinition of the class loader, like the one we describe in Section 5, can operate on the LIME tuple space associated to the agent for which the class needs to be resolved, and query it using the operations provided by LIME. Thus, the class loading mechanism can now resolve class names by leveraging off of the federated code base to retrieve and dynamically link classes in a location transparent fashion, e.g., through a rd, or use location parameters to narrow the scope of searches, e.g, down to a given host or agent.

Nevertheless, the use of transiently shared tuple spaces needs not be confined to the innards of the class loading mechanism. The coordinated agents can be allowed to manipulate directly the tuple spaces holding classes, and representing code bases. In this case, the potential of combining transiently shared tuple spaces and mobile code is fully available for coordination, as LIME operations on the transiently shared tuple space are now available to manipulate the federated code base. Hence, not only can an agent proactively query up to the whole system for a given class, but it can also insert a class tuple into the code base of another agent by using the out[λ] operation, with the semantics of engagement and misplaced tuples even taking care of disconnection and subsequent reconciliation of the federated code base. This new class can then be used by the receiving agent to execute tasks in previously unknown ways, or even to behave according to a new coordination protocol. Blocking operations acquire new uses, allowing agents to synchronize not only on the presence of data needed by the computation, but also on the presence of code needed to perform, or augment, the computation itself. LIME reactive operations add even more degrees of freedom, by allowing agents to monitor the federated code base and react to changes with different atomicity guarantees. Reactions can be exploited straightforwardly to monitor the federated code base for new versions of relevant classes. Replication schemes can be implemented where a new class in an agent's code base is immediately replicated into the code base of all the other agents. The content of an agent's code base can be monitored to be aware of the current "skills" of the agent. The possibilities become endless.

Essentially, by exploiting the notion of transiently shared tuple space for code mobility we are defining an enhanced coordination approach that, besides accommodating reconfiguration due to mobility and providing various degrees of location transparency, enables a new form of coordination no longer limited to data exchange, but encompassing also the exchange of fragments of behavior.

5 Enhancing a Mobile Code Toolkit with Location Transparent Class Loading

To understand what it takes to bring transiently shared tuple spaces into an existing mobile code system, we implemented a proof-of-concept prototype, whose

design and implementation[3] is the subject of this section. The prototype extends the functionality of an existing mobile code toolkit by coupling its class loading mechanism with a transiently shared tuple space, and by identifying an appropriate interface for managing the resulting federated code base.

The toolkit we chose to extend is called μCODE [14], and is available at `http://mucode.sourceforge.net`. The availability of the toolkit as open source was one of the factors driving the choice, together with more pragmatic reasons like the expertise of one of the authors as a developer for both LIME and μCODE, and the fact that the two systems have already been shown to work seamlessly together. Nevertheless, the approach followed here can most likely be adapted to other systems, as discussed later in this section.

We now review briefly the salient characteristics of μCODE, and then discuss how transiently shared tuple spaces holding classes have been integrated into it.

5.1 μCode

μCODE [14] is a lightweight and flexible toolkit for code mobility that, in contrast with most of similar platforms, strives for minimality and places a lot of emphasis on modularity. μCODE revolves around three fundamental concepts: groups, group handlers, and class spaces.

Groups are the unit of mobility, and provide a container that can be filled with arbitrary classes and objects (including thread objects) and shipped to a destination. Classes and objects need not belong to the same thread. Moreover, the programmer may choose to insert in the group only some of the classes needed at the destination, and let the system exploit remote dynamic linking for downloading the missing classes from a target specified at group creation time.

The destination of a group is a μServer, an abstraction of the run-time support. In the destination μServer, the mix of classes and objects must be extracted from the group and used in some coherent way, possibly to generate new threads. This is the task of the *group handler*, an object specified by the programmer at group creation time, which is instantiated in the destination μServer where its operations are automatically invoked. Any object can be a group handler. Programmers can define their own specialized group handlers and, doing so, define their own mobility primitives.

During group reconstruction, the system needs to locate classes and make them available to the group handler. The classes extracted from the group must be placed into a separate name space, to avoid name clashes with classes reconstructed from other groups. This capability is provided by the *class space*. Classes shipped in the same group are placed together in a private class space, associated with that group. However, these classes can later be "published" in a shared class space associated to a μServer, where they become available to all the threads executing in it, as well as to remote ones.

Class spaces play also a role in the resolution of class names. When a class name C needs to be resolved during execution of a thread t managed by a μServer

[3] The implementation is included in the public distributions of LIME and μCODE.

S, the redefined class loader of μCODE is invoked to search for C's bytecode by performing the following steps: *i)* check whether C is a *ubiquitous class*, i.e. a class available on every μServer (e.g., system classes); *ii)* search for C in the private class space associated with t in S; *iii)* search for C in the shared class space associated with S; *iv)* if t is allowed to perform dynamic download, retrieve C from the remote μServer specified by the user at migration time, and load C; *v)* if C cannot be found, throw a `ClassNotFoundException`.

Moreover, μCODE provides higher-level abstractions built on the core concepts defined thus far. These abstractions include primitives to remotely clone and spawn threads, ship and fetch classes to and from a remote μServer, and a full-fledged implementation of the mobile agent concept.

5.2 Providing Transiently Shared Class Spaces in μCode

We now discuss in detail the design of our prototype, by highlighting the design choices we made, and the extensions that were required to μCODE. Notably, no modification was required to LIME.

Generalizing the Addressing of Dynamic Link Sources. μCODE supports dynamic linking using a traditional scheme where the address `host:port` of the μServer holding the class is somehow known. Hence, we need to change this address format into one supporting a location transparent scheme. In our prototype, this is achieved by using Uniform Resource Identifiers (URI) [1] instead of arbitrary strings, thus effectively generalizing the way the dynamic link source is specified. The old μCODE format becomes now a URI `mucode://host:port`, while a URI starting with `lime://`, together with the appropriate addressing scheme we describe below, exploits LIME for identifying a location transparent link source, or an appropriate portion of the federated code base.

Incidentally, this effectively decouples the mechanism used to retrieve the missing classes, and thus opens up additional possibilities for dynamic class loading, e.g., loading it from an HTTP connection, or through coordination infrastructures other than LIME.

Generalizing μCODE Class Spaces. Class spaces were introduced in μCODE with the intent of providing the programmer with more flexibility in dealing directly with mobile code, inspired by considerations similar to those presented in Section 2 and 4. By leveraging off transiently shared tuple spaces, we essentially provide a generalization of the class space concept by stretching its boundaries to cover potentially the whole system.

The private class space is unchanged in our prototype, as it is necessary for providing a separate name space for loading classes during group reconstruction. Instead, the shared class space is now defined by a LIME transiently shared tuple space whose tuples may contain classes, in bytecode form, and hence represents a sort of *class tuple space*. By borrowing ideas from the LIME model, we can define the following kinds of class tuple spaces, capturing varying degrees of sharing:

– *Agent class tuple space* (`lime://<lime agent id>/<tuple space name>`). Classes are searched only in the class tuple space of the agent whose LIME identifier is provided. Note that it is different from the private class space, that is not a tuple space and remains hidden from the rest of the system.
– *μServer class tuple space* (`lime://<lime host id>/<tuple space name>`). Classes are searched only in the transiently shared tuple space generated by the agents currently hosted by the μServer.
– *Federated class tuple space* (`lime://*/<tuple space name>`). Classes are searched in the whole transiently shared tuple space.

In the classification above, the format of the URI for each kind of class tuple space is provided. This address can be used to restrict the scope of searches when a class resolution needs to retrieve a missing class through remote dynamic linking. Alternatively, applications may query and manipulate the class tuple space by using LIME operations, by treating class tuples as normal data tuples.

Storing Classes in a μServer. The shared class space of μCODE provided a way to "publish" to the rest of the system some of the classes belonging to a group, and originally in its private space. We now provide this functionality in a much more powerful fashion by using transiently shared tuple spaces.

Nevertheless, shared class spaces also provided a persistency root for classes that was useful in several situations, e.g., to implement class caching schemes. This capability would now be lost, due to the transient nature of the shared tuple space containing the classes: only those classes associated to a running LIME agent would remain available. The solution is to associate a transiently shared tuple space to the μServer. The semantics of LIME forces the `LimeTupleSpace` object representing a transiently shared tuple space to be permanently and exclusively associated to the agent that created it. The μServer must then become also a LIME agent. This is achieved straightforwardly by letting the class `MuServer` implement the `lime.ILimeAgent` interface.

Placing Classes into Tuples. In our scheme, mobile code is transferred by retrieving a tuple containing the class bytecode. A design issue is then how to embed code into a tuple. A straightforward solution is simply to place the byte array containing the bytecode in one of the tuple fields, and use the others to provide information used to match the tuple (i.e., at least the class name), like in `<String name, byte[] bc>`[4]. Different applications may associate different information to the class bytecode, e.g., a version number, certificates, or application-dependent information, and hence define different tuple formats.

Redefining the Class Loader. Last but not least, the loading strategy of the class loader embedded in μCODE must be changed to encompass searches in the transiently shared tuple space.

[4] The API of the prototype actually defines a `MarshalledClass` helper class, to overcome the Java limitation about using scalar types (like `byte[]`) in tuple fields.

```
public class DistributedClassSpace extends ClassSpace {
  public boolean containsClass(String className, String uri);
  public void removeClass(String className, String uri);
  public Class getClass(String className, String uri);
  public void getClassByteCode(String className, String uri);
  public void putClassByteCode(String className, String uri);
  public int addClassListener(String name, String uri,
                              ClassListener listener, short mode);
  public void removeClassListener(int id);
}
```

Fig. 2. The class `DistributedClassSpace`. Exception declarations and overloaded methods are omitted for the sake of clarity.

The strategy we adopted is similar to the original one, in that ubiquitous classes are searched first, followed by classes in the private space. Nevertheless, the steps of searching into the shared class space and possibly attempting a remote dynamic linking are now collapsed into a single one. In fact, the shared class space is now stretched to encompass possibly the whole system, and the dynamic link source parameter specifies a scope for searching the class that ranges from the class tuple space of a single agent, finer-grained than the original shared class space, up to the whole federated tuple space[5].

The actual implementation of class retrieval is straightforward, and relies on parsing the URI to determine the parameters to be passed to the LIME operations performing the actual query on the class tuple space.

5.3 Hiding LIME

As we mentioned earlier, the possibility for coordinated agents to access directly the tuple spaces containing the classes is the alternative that leaves more degrees of freedom to the application, and that leverages the most of the uniform access provided by the coordination infrastructure to the application code and state.

Nevertheless, for other applications it might be reasonable to shield the coordination infrastructure behind a set of interfaces that hide the details of how classes are retrieved. This latter alternative is surely more constrained. On the other hand, it decouples the API used to access the federated code base from the coordination infrastructure that enables it.

In our prototype, we leveraged off of the class space concept already provided by μCODE. The class `DistributedClassSpace`, shown in Figure 2, specializes `mucode.ClassSpace` and redefines the methods that originally allowed to query the local class space (e.g., `getClass`) to perform the same query on the federated

[5] Searches can also be restricted to the hosting μServer (i.e., without considering the spaces of co-located agents), by specifying a URI containing the LIME agent identifier associated to the μServer.

code base using the URI scheme we defined previously. This is the class that holds a reference to the actual transiently shared tuple space containing the code base.

Moreover, `DistributedClassSpace` is equipped also with the ability to react to the insertion of a class in the class space. This is accomplished by registering a class listener, implemented using a LIME (weak) reaction. The `mode` parameter allows to specify whether the listener should fire only once, or remain registered to detect the appearance of other classes, until explicitly deregistered.

5.4 Other Considerations

The approach we followed for coupling transiently shared tuple spaces with μCODE was to specialize some of μCODE classes. The task was simplified by the fact that all of the relevant classes, including the class loader, were already publicly accessible—a condition unlikely to hold true in the majority of mobile code systems. Nevertheless, a small number of minor changes, e.g., to allow access to class features originally declared as `final`, indeed required access to the source code.

In principle, the approach we followed can be applied also to other systems, e.g., mobile agent platforms. Several are available, and many are also open source. For instance, this is the case of the Aglets system, whose latest release actually provides a `CacheManager` class as part of the run-time support, that could be exposed to the user to provide functionality similar to μCODE class spaces, and extended with a design similar to the one described in Section 5.3. For applications that do not rely on mobile agents, a custom class loader containing an application-specific loading strategy could also be developed.

As for LIME, no modification was required but a couple of points need further elaboration. First, we relied on an extension of LIME, contained in the current public distribution, that provides the ability to invoke probe operations like `inp` and `rdp` on the whole federated tuple space. In the original LIME model, the use of probes was limited only to the transiently shared tuple space of a given host or agent, in the attempt to retain the atomicity guarantees of probe invocation and yet allow for a practical and efficient implementation. The aforementioned extension strikes a different balance between the two aspects, by lifting the atomicity requirement. Hence, a probe may now execute on the whole federated tuple space, but its (distributed) execution is not atomic, and is then allowed to miss some tuples that might have appeared in the tuple space while the probe was executing. This extension is actually the initial step of a broader ongoing effort by the authors of LIME to weaken the atomicity requirements of the model.

Moreover, the LIME implementation currently requires that threads accessing a tuple space must implement an `ILimeAgent` interface. This means that application threads that want to exploit the mechanisms described here must comply to this requirement, e.g., by subclassing from `StationaryAgent` or `MobileAgent`, or by directly implementing the interface. This is not likely to be a big obstacle for most applications, and surely it was not for the simple applications we used to test our proof-of-concept prototype. However, further experience with our ap-

proach may lead us in a different direction, e.g., requiring modifications to the LIME implementation to lift this constraint during accesses to a class loader.

6 Discussion and Related Work

The idea of placing code in a tuple space is already present in the Linda model, where the eval primitive allows for a code fragment to be inserted in the tuple space, and eventually evaluated to produce a new tuple. Other approaches, e.g., PoliS [5] allow tuple spaces to contain rules that, when fired, can modify the tuple space. However, these and other approaches enhance the coordination infrastructure by introducing behavior into it, under the form of tuples containing some form of *"active"* code that is able to modify the tuple space by itself.

Instead, our work takes a different viewpoint where the tuples contain *"passive"* code that cannot be activated by itself and modify the tuple space, rather it is exploited directly by the applications, or indirectly through the class loading mechanism, to augment the functionality of the coordinated agents. Interestingly, in principle the two perspectives are complementary and could be rejoined. Thus, for instance, it could be conceivable to have eval tuples or PoliS rules contain only a subset of the code specifying the active behavior, and let the missing portion be linked dynamically into the tuple by retrieving it from the current content of the tuple space, be it transiently (like in LIME) or permanently shared (like in Linda, PoliS, and other existing approaches).

Existing Java-based tuple space systems surprisingly do not leverage much of mobile code, although they allow Java objects to be contained in tuple fields. For instance, both TSpaces [10] and JavaSpaces [17] allow the coordinated agents to access a remote tuple space using a client-server paradigm. However, TSpaces assumes that the classes for the objects contained in a tuple being transferred between an agent and the tuple space server are found at the latter, thus essentially relying on the default loading strategy of Java[6]. Instead, JavaSpaces provides a more elaborate scheme that is nonetheless location-based. Classes are annotated with the URL of a codebase, usually constituted by a Web server hosted by each JavaSpaces client. This latter mechanism is exploited by Jini [18] to implement a discovery and lookup service, by storing in the tuple space objects representing service proxies. Upon a service request, a reference to these service objects is passed to the client, which can access the service remotely. Since this mechanism is implemented on top of RMI, dynamic linking is triggered when some of the stubs needed for remote invocation are missing on either the client or the server.

With our approach, we are able to provide a similar functionality—and more. First of all, we are able to relocate code separately from objects, and also independently from the immediate need of remote dynamic linking due to name resolution. Moreover, transiently shared spaces eliminate the need to know the location of a lookup server, by allowing queries that are intrinsically location transparent. Actually, the peer-to-peer perspective adopted by LIME, opposed

[6] A location-based scheme similar to the one described later for JavaSpaces was posted in the TSpaces mailing list, but apparently never made it into the official distribution.

to the client-server architecture exploited by Jini, allows any client to host its own code base and services without the need to register with a lookup server or to rely on a centralized code base, and thus yields an architecture that is inherently more amenable to reconfiguration.

In this presentation, we did not consider security issues, since our focus is in examining the potential of the novel idea of coupling transiently shared tuple spaces and mobile code. In our approach security issues are potentially exacerbated by the fact that now agents are allowed to manipulate each others' code base, and can potentially do this on a system-wide basis. Nevertheless, the mechanisms commonly used for sandboxing in Java-based systems, coupled with well-known techniques based on certificates and public keys should cover most of the needs. Moreover, recent proposals of secure extensions for LIME [4] or other tuple space models [2] are likely to be adaptable to our approach.

Similarly, in this paper we did not address scalability issues. The ability to issue class retrieval requests potentially spanning the whole system surely adds expressive power, but may lead to poor performance. Again, we expect these problems to be dealt with at another level, i.e., by modifying the underlying model and system supporting transiently shared tuple spaces, LIME in this case. As we briefly mentioned earlier, ongoing work by the authors of LIME and by other researchers (e.g., [4, 3]) aims at lifting some of the atomicity assumptions of the model, in an attempt to provide a more efficient and scalable implementation.

7 Conclusions and Future Work

Currently available systems do not explore fully the potential of code mobility, in that they support linking schemes where code is dynamically retrieved from a well-known location, and they do not expose the code base to applications. In general, the expressiveness of the abstractions currently available to deal with mobile code is poor if compared to the potential that code mobility may unleash.

In this paper, we presented a different perspective by coupling the idea of transiently shared tuple space, originally introduced in the LIME model, with the mechanisms supporting mobile code. The result is an enhanced model of coordination where the code base, that is indeed exposed to applications, is virtually shared across the whole system and where mobile code is dealt with both in a location aware and location transparent fashion. The fact that code can be treated as data enables the use of the existing coordination primitives and supports a unified treatment of the code and state necessary to coordination.

Future work on this topic will focus on the development of applications and middleware based on the concepts described in this paper, to assess the implications of this model. In particular, an ongoing effort is currently aiming at implementing a Jini-like discovery and lookup service that leverages off of the peculiarity of our coordination infrastructure.

Acknowledgments The authors wish to thank Amy Murphy for her insightful comments on an early draft of this paper.

References

1. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF Network Working Group, RFC 2396, August 1998.
2. C. Bryce, M. Oriol, and J. Vitek. Secure object spaces: A coordination model for agents. In *Proc. of COORDINATION'99*, LNCS 1594, pages 4–20. Springer, 1999.
3. N. Busi and G. Zavattaro. Some Thoughts on Transiently Shared Tuple Spaces. In *Proc. of Workshop on Software Engineering and Mobility, co-located with the 23rd Int. Conf. on Software Engineering (ICSE01)*, May 2001.
4. B. Carbunar, M.T. Valente, and J. Vitek. Lime Revisited—Reverse Engineering an Agent Communication Model. In *Proc. of the 5th Int. Conf. on Mobile Agents (MA 2001)*, LNCS 2240, pages 54–69. Springer, December 2001.
5. P. Ciancarini, F. Franzè, and C. Mascolo. Using a Coordination Language to Specify and Analyze Systems Containing Mobile Components. *ACM Trans. on Software Engineering and Methodology*, 9(2):167–198, April 2000.
6. Ad Astra Engineering. Jumping Beans Web page. `http://www.jumpingbeans.com`.
7. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
8. D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
9. C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 25–47. Springer, April 1997.
10. IBM Research. TSpaces Web page. `http://www.almaden.ibm.com/cs/TSpaces`.
11. D.B. Lange and M. Oshima, editors. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
12. M.Corson, J.Macker, and G.Cinciarone. Internet-Based Mobile Ad Hoc Networking. *Internet Computing*, 3(4), 1999.
13. A.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.
14. G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of Mobile Agents: 2nd Int. Workshop MA'98*, LNCS 1477, pages 160–171. Springer, September 1998.
15. G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *Proc. of the 21st Int. Conf. on Software Engineering (ICSE'99)*, pages 368–377, May 1999.
16. M. Straßer, J. Baumann, and F. Hohl. Mole—A Java Based Mobile Agent System. In M. Mühlaüser, editor, *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conf. on Object-Oriented Programming ECOOP'96*, pages 327–334. dpunkt, July 1996.
17. Sun Microsystems. JavaSpaces Web page. `http://java.sun.com/products/javaspaces`.
18. Sun Microsystems. Jini Web page. `http://www.sun.com/jini`.