LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications

Gian Pietro Picco, Davide Balzarotti and Paolo Costa
Dip. di Elettronica e Informazione, Politecnico di Milano, Italy
{picco, balzarot, costa}@elet.polimi.it

ABSTRACT

The tuple space model inspired by Linda has recently been rediscovered by distributed middleware. Moreover, some researchers also applied it in the challenging scenarios involving mobility and more specifically context-aware computing. Context information can be stored in the tuple space, and queried like any other data.

Nevertheless, it turns out that conventional tuple space implementations fall short of expectations in this new domain. On one hand, many of the available systems provide a wealth of features, which make the resulting implementation unnecessarily bloated and incompatible with the tight resource constraints typical of this field. Moreover, the traditional Linda matching semantics, based on value equality are not appropriate for context-aware computing, where queries are often formulated over value ranges, and where there is a prominent need to deal with imprecise information coming from multiple sources.

In this paper, we describe a new tuple space implementation called LighTS. Originally developed as the tuple space core of the Lime [11] system, LighTS provides an extensible framework that makes it easy to introduce extensions to the tuple space, and in general customize the tuple space implementation. The design and programming interface of LighTS is presented, and its flexibility demonstrated by illustrating extensions that proved useful in the development of context-aware applications.

1. INTRODUCTION

The tuple space model inspired by Linda [8] has recently been rediscovered by distributed middleware. Commercial systems (e.g., TSpaces [1], JavaSpaces [2], GigaSpaces [3]) as well as academic ones (e.g., MARS [6], TuCSoN [13], Klaim [12], Lime [11]) are currently available.

Among these, the Lime system is particularly interesting, in that it adapts and extends Linda towards mobility, by transforming its global and persistent tuple space into one that is federated and transiently shared according to connectivity. Moreover, this system has recently been used to develop context-aware applications.

In [10], the authors describe a simple 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 displaying of the current location and/or trajectory of any other user, provided wireless connectivity is available towards her. The application is implemented by exploiting tuple spaces as repositories for context information—i.e., location data. The primitives of Lime are then used to seamlessly perform queries not only on a local tuple space, but on all the spaces in range. For instance, the location of a user can be determined by performing a read operation for the location tuple associated to the given user identifier. The "lesson learned" distilled from this work is 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 coordination primitives—for accessing both forms of data, therefore simplifying the programmer's chore.

Nevertheless, as the authors discuss in their findings, it turns out that the traditional matching semantics of Linda, based on comparing the exact values of tuple fields, is insufficient for the needs of context-aware applications. Indeed, context-aware queries rarely revolve around exact values. For instance, in a sensor monitoring application, it may be required to find the identifiers of all the temperature sensors registering a value between 20 and 25 degrees. Or, in the application of [10] it may be needed to find the users within 500m, or those within r meters from the point (x, y). Often, even these queries are too precise, in that the user may have enough information only to formulate requests as informal as "find the sensors recording a hot temperature", or "find the users close to me". These needs sometimes surface also in conventional applications, but they are definitely exacerbated in context-aware ones.

In this paper, we set out to close the gap between the tuple space model and context-aware applications, by extending LighTS, the tuple space engine at the core of the Lime system. In contrast with available tuple space systems, LighTS [4] was designed to be extremely lightweight and minimal, by providing a local tuple space with support

SAC'05, March 13-17, 2005, Santa Fe, New Mexico, USA.

only for the basic Linda operations. Indeed, distribution, event notification, transactions—features typically provided by other systems—are built on top of LighTS by Lime. Nevertheless, the lack of sophisticated features is compensated by a design that renders LighTS highly customizable and extensible. In essence, the tuple space abstraction provided by LighTS was conceived as a *framework* (in the object-oriented sense) rather than a closed *system*. The built-in instantiation of such framework provides the traditional Linda abstractions, similarly to many other systems. At the same time, however, the modularity and encapsulation provided by its object-oriented design leaves room for customization, empowering the programmer with the ability to easily change performance aspects (e.g., changing the tuple space engine) or semantics features (e.g., redefine matching rules or add new features). This flexibility and extensibility, together its small footprint and simple design, are the defining features of LighTS.

We put forth two contributions. First, we present the overall architecture and programming interface of LighTS, and describe the mechanisms supporting customization and extension. Second, we show how these mechanisms can be exploited straightforwardly to suit the needs of context-aware applications. In particular, we describe the design of two additional matching semantics (one based on comparison over value ranges and one based on fuzzy logic), and of a new feature which enables data aggregation at the tuple level. Finally, the ability to change the back-end of the implementation enables us to provide different deployment alternatives, including support for devices running Java2 Micro Edition (J2ME).

The paper is organized as follows. Section 2 is a concise overview of Linda. Section 3 presents the overall design of LighTS, and shows how the resulting framework can be easily extended both in terms of performance and semantics. Section 4 discusses the extensions we introduced to better suit the requirements posed by context-aware applications. Section 5 briefly reports about implementation details and availability of the software package. Finally, Section 6 ends the paper with brief concluding remarks.

2. LINDA IN A NUTSHELL

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

Tuples are added to a tuple space by performing an **out**(t) operation, and can be removed by executing **in**(p). Tuples are anonymous, thus their selection takes place through pattern matching on the tuple content. The argument p is often called a *template* or *pattern*, and its fields contain either *actuals* or *formals*. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of ⟨"foo", ?integer, ?float⟩ are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by **in** is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive **rd**(p) operation. Both **in** and **rd** are blocking, i.e., if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple becomes available. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives **inp** and **rdp**, called *probes*, that allow non-blocking access to the tuple space. Moreover, some variants of Linda (e.g., [14]) provide also *bulk operations*, which can be used to retrieve all matching tuples in one step[1].

3. THE DESIGN OF LIGHTS

In this section we present the core features of LighTS, followed by the mechanisms for customizing and extending the framework, which are exploited in Section 4 to build new features useful for context-aware applications.

3.1 The Core API of LighTS

The core of LighTS is contained in the `lights` package and is constituted by a set of interfaces that model the fundamental concepts of Linda (i.e., tuple spaces, tuples, and fields) and by a built-in implementation of these interfaces.

Tuple spaces. Figure 1 shows[2] the interface `ITupleSpace`, which must be implemented by every tuple space object. The interface contains the basic Linda operations described in Section 2, i.e., insertion (`out`), blocking queries (`in`, `rd`), probes (`inp`, `rdp`), and bulk operations (`outg`, `ing`, `rdg`). Tuple spaces are expected to be created with a name, enabling an application to manage multiple tuple spaces, as suggested in [7]. The name of a tuple space can be retrieved through the method `getName`. Finally, `ITupleSpace` provides also a method `count` that returns the number of tuples currently in the tuple space.

Being an interface, `ITupleSpace` specifies only a syntactic contract between the implementor and the user of the implementing object, and nothing can be said about the semantics of the actual implementation. Therefore, for instance it is not possible to prescribe that accesses to the tuple space must be mutually exclusive, as usually required by Linda. This is an intrinsic limitation in expressivenes of the Java language (and other object-oriented approaches). Nevertheless, the built-in `TupleSpace` class, which implements `ITupleSpace`, behaves like a traditional Linda tuple space by preserving atomicity of operations. Moreover, tuple insertion is performed by introducing in the tuple space a *copy* of the `tuple` parameter, to prevent side effects through aliasing. Since tuples may contain complex objects, copying relies on the semantics of Java serialization, which already deals with aliases inside object graphs. Upon insertion, a deep copy of the `tuple` parameter is obtained through serialization and immediate deserialization. A similar process is performed when a non-destructive read operation (`rd`, `rdp`, or `rdg`) is performed. Nevertheless, our `TupleSpace` implementation can be configured to reduce the impact of serialization and trade space for speed, by storing a copy of the byte array containing the serialized tuple together with the tuple itself. This way, read operations are faster since they need to perform only a deserialization step to return their result. The desired configuration is specified at creation time

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

```
public interface ITupleSpace {
  String getName();
  void out(ITuple tuple);
  void outg(ITuple[] tuples);
  ITuple in(ITuple template);
  ITuple inp(ITuple template);
  ITuple[] ing(ITuple template);
  ITuple rd(ITuple template);
  ITuple rdp(ITuple template);
  ITuple[] rdg(ITuple template);
  int count(ITuple template);
}
public interface ITuple {
  ITuple add(IField field);
  ITuple set(IField field, int index);
  IField get(int index);
  ITuple insertAt(IField field, int index);
  ITuple removeAt(int index);
  IField[] getFields();
  int length();
  boolean matches(ITuple tuple);
}
public interface IField {
  Class getType();
  IField setType(Class classObj);
  boolean matches(IField field);
}
public interface IValuedField extends IField {
  boolean isFormal();
  java.io.Serializable getValue();
  IValuedField setValue(java.io.Serializable obj);
}
```

Figure 1: The core interfaces of LighTS.

through the constructor, which also enables setting the name of the tuple space.

Tuples. Figure 1 shows the interface `ITuple`, which provides methods for manipulating tuples. A field at a given position in the tuple (from 0 to `length()-1`) can be read (`get`), changed (`set`), or removed (`removeAt`). A new field can be appended at the end of the tuple (`add`), as well as at any other position (`insertAt`). The fields composing the tuple can also be read collectively into an array (`getFields`). No syntactic distinction is made between tuples and templates—they are both `ITuple` objects.

The key functionality, however, is provided by the `matches` method, which is expected to embody the rules governing tuple matching and therefore is the one whose redefinition enables alternative semantics. This method is assumed to be automatically invoked by the run-time whenever a match must be resolved, and to proceed by comparing the tuple object on which `matches` is invoked—behaving as a template—against the tuple passed as a parameter. By virtue of encapsulation, the matching rule implemented in `matches` is entirely dependent on the template's class, implementing `ITuple`. Nevertheless, by virtue of polymorphism and dynamic typing, the behavior of the run-time is the same regardless of the details of the matching rule, since the only assumption it makes is to operate on a template implementing `ITuple`.

The default semantics of `matches` as implemented in the built-in `Tuple` is the traditional one. When `matches` is invoked on a template against a parameter `tuple` it returns `true` if:

1. the template and the tuple have the same arity, and
2. the i^{th} template field matches the i^{th} tuple field.

Field matching is described next.

Fields. Figure 1 shows the interfaces representing tuple fields. `IField` provides the minimal abstraction of a *typed* tuple field. Methods are provided for accessing the field's type (`getType`, `setType`). As with `ITuple`, `IField` contains a method `matches`, where the implementing classes specify the matching semantics, as exemplified later on.

The features of `IField` are enough to represent a *formal* but not an *actual* field, in that there is no notion of a field's value. This abstraction is provided by the interface `IValuedField` which extends `IField` with the accessors necessary to deal with the value (`getValue`, `setValue`), as well as with a way to test whether the current field is a formal (`isFormal`). Note that `setValue` accepts any `Object` as a parameter. Moreover, the field's type is automatically set to the parameter's class.

The need for two separate interfaces is not immediately evident if one considers only the pragmatic need of supporting the basic Linda operations. As a matter of fact, the built-in `Field` implements both interfaces. However, this separation provides a cleaner decoupling when matching semantics that do not rely on exact value match are considered, as in the examples we provide later in this and the next section. The built-in `Field` is defined so that `this.matches(f)` returns `true` if:

1. `this` and `f` have the same type;
2. if `this` and `f` are both actuals (i.e., `isFormal()` returns `false` for both of them) they also have the same value.

Equality of types and values is resolved by relying on the `equals` method—as done usually in Java. Also, a field that does not implement `IValuedField` is automatically considered a formal.

Programming example. Let us walk through the simple task of inserting two tuples in a tuple space and retrieving one of them. We assume a statement `import lights.*` has been specified. First, we need to create a tuple space:

```
ITupleSpace ts = new TupleSpace("SAC05");
```

Then, we need to create the two tuples. Fields can be created as:

```
IField f1 = new Field().setValue("Paolo");
IField f2 = new Field().setValue(new Integer(10));
```

and then assembled in a tuple:

```
ITuple t1 = new Tuple().add(f1);
t1.add(f2);
```

In alternative, we can leverage of the fact that `ITuple` methods always return an `ITuple` object (although not strictly necessary from a purely semantic standpoint) and combine multiple statements in a single one:

```
ITuple t2 = new Tuple()
            .add(new Field().setValue("Davide"))
            .add(new Field().setValue(new Integer(20));
```

The tuples can be inserted one at a time, or together in a single atomic step, as in:

```
ts.outg(new ITuple[] = {t1, t2});
```

Templates are created just like tuples:

```
ITuple p = new Tuple()
           .add(new Field().setType(String.class)
           .add(new Field().setValue(new Integer(10));
```

Finally, the probe operation

```
ITuple result = ts.rdp(p);
```

will return a copy of the first tuple in `result`. More examples are available at [4] and [5].

3.2 Customizing LighTS

The LighTS framework is designed to provide the minimal set of features implementing a Linda-like tuple space and, at the same time, to offer the necessary building blocks for customizing and extending it. We now discuss the most relevant customization opportunities, some of which are actually exploited in the additional packages included in the LighTS distribution.

Changing the tuple space engine. The tuple space implementation in the `lights` core package is very simple[3]. Notably, the data structure holding tuples is simply an in-memory `java.util.Vector` object, which is scanned linearly upon a query operation. This design is motivated by the need to support deployment on resource-constrained devices—a requirement of the Lime project—and admittedly may not perform reasonably in other scenarios.

Nevertheless, the information hiding provided by the core interfaces greatly simplifies the task of realizing more sophisticated implementations (e.g., providing persistence, checkpointing, or more scalable matching algorithms), with little or no impact on the application code. At one extreme, one could even sneak a commercial system (e.g., TSpaces or GigaSpaces) behind the LighTS interfaces, e.g., to enable the development of applications that can be deployed on top of different tuple spaces engines. In a research context, this is particularly useful to evaluate different alternatives without the need to fully rewrite the application.

To make this development strategy easier, the `lights.adapters` package provides the building blocks necessary to substitute the built-in implementation in `lights` with a different one. The classes `TupleSpace`, `Tuple`, and `Field` in such package provide wrappers that on one hand implement the required `lights` interfaces, and on the other contain an adapter object implementing the required functionality, and to which interface operations are delegated[4]. The abstract class `TupleSpaceFactory`, to be derived by the actual adaptation package, enables selection of the appropriate set of adapter classes at start-up. To illustrate these features, an adapter for TSpaces is included in the current LighTS distribution, and one for GigaSpaces is being finalized. Also, a tuple space adapter for J2ME has been implemented, which again confirms not only the versatility of the framework, but also that its inherent simplicity eases its deployment even on devices with tight resource constraints.

Changing the matching semantics. Tuple space systems vary considerable in terms of their matching semantics. For instance, TSpaces enables the use of subtyping rules in matching field types, and relies on the (re)definition of the `equals` method for matching field values. Instead, JavaSpaces matches two fields by comparing their serialized forms. Also, a JavaSpaces tuple (or *entry* in Sun's jargon) is represented by a class, and therefore subtyping rules among tuples take part in matching. In TSpaces, this is enabled only if tuples are derived from a specific root class, otherwise it is not allowed by default `Tuple` class. Finally, TSpaces requires two matching tuples to have the same arity, while JavaSpaces lifts this constraint when a tuple is a subtype of another. This short comparison evidences that several variations are possible, with tradeoffs in expressiveness, ease of use, and integration with object-orientation. As a consequence, committing to a particular choice may end up hampering development of some applications.

LighTS was designed since the beginning with this problem in mind. The default matching in LighTS relies on the `equals` method, disallows field or tuple subtyping, and requires equal tuple arity. Nevertheless, the `lights.extensions` package contains several examples that show how easy it is to provide alternative semantics, by exploiting interfaces and other aspects of our object-oriented design.

The class `SubtypeField`, for instance, takes subtype compatibility into account during field matching. Providing this feature is as simple as subclassing `lights.Field` and redefining `matches` by including the additional constraint

```
getType().isAssignableFrom(field.getType)
```

where `field` is the input parameter of `matches`. Similarly, matching based on inequality is trivially provided by `NotEqualField`.

Tuple matching can be similarly redefined. `PrefixTuple` extends `Tuple` by allowing a template of arity l to return a successful match against any tuple whose first l fields match, in order, with the template ones—a need that often arises in practice in tuple space based applications. Incidentally, this also provides a straightforward way to retrieve all tuples in the tuple space. Again, the only change required is in the implementation of `matches`.

Modifications can be more complex. For instance, the same package contains also a `RegexField` that allows matching of string fields using regular expressions and requires additional attributes and methods for setting and compiling the expression using the Java library. Many other extensions are possible. One could easily implement SQL- or XML-based matching, and many others. Thus far, development of extensions has been driven by pragmatic needs that arose during experiences with the Lime middleware. In the next section, we illustrate some extensions we found useful in developing context-aware applications.

4. SUPPORTING CONTEXT-AWARENESS

As we discussed in Section 1, the tuple space abstraction is particularly well-suited for context-awareness. Context data can be stored in the tuple space, and made accessible by leveraging of the nice decoupling properties of the Linda approach. Nevertheless, the standard matching based on exact values is largely insufficient for context-aware applications. Here, we describe two alternative matching semantics, and also show how to exploit tuples in a way that supports data aggregation—another relevant concern in context-aware computing.

4.1 Matching on Value Ranges

In context-aware applications, many queries require to determine whether a given value from contextual data (e.g., temperature from a sensor) is within an allowed range (e.g., 35-38°C). This capability is not easily built on top of a conventional system that provides only exact value matching, and requires considerable programming and computing effort. For instance, a common hack is to retrieve tuples by matching on the other fields, and explicitly code in the application the matching on the field involving a value range.

LighTS overcomes this limitation by leveraging the mechanisms for extension we illustrated in the previous section. The class `RangeField` in the `extensions` package provides

[3]Space limitations force us to redirect the reader looking for more details to the online documentation and source [4].
[4]Extensions are currently not supported by adapters.

```
public class RangeField extends TypedField {
  public RangeField()
  public RangeField setLowerBound(Comparable low,
                                  boolean included)
  public RangeField setUpperBound(Comparable up,
                                  boolean included)
  public Comparable getLowerBound()
  public Comparable getUpperBound()
  public boolean isLowerBoundIncluded()
  public boolean isUpperBoundIncluded()
  public boolean matches(IField field)
}
```

Figure 2: The class `RangeField`.

methods for specifying the lower and upper bounds of the value range and whether they are included in it, as shown in Figure 2. The snippet below shows how to match over the aforementioned temperature range, without including the lower bound of 35°C:

```
RangeField rf = new RangeField()
                .setLowerBound(new Float(35), false)
                .setUpperBound(new Float(38), true);
ITuple result = tuplespace.rdp(new Tuple().add(rf));
```

Bounds can be represented by any object implementing the interface `java.lang.Comparable`. `RangeField` extends `lights.extensions.TypedField`—a convenience abstract class that serves the only purpose of implementing the `IField` interface—by simply adding attributes holding information about bound values and redefining `matches` with the trivial constraint necessary to check that the field being compared against falls in the required range. As the reader can see, the extent of modifications necessary to implement the required semantics is minimal and extremely simple, while the impact on expressiveness is remarkable.

4.2 Fuzzy Matching: Dealing with Uncertainty

In several applications the power of range matching is not enough, as users may not have the knowledge required to formulate precise queries. For instance, a user may request to find a restaurant that is to *close* to her, without bothering about estimating a reasonable range based on the urban density of the surrounding area. Indeed, people commonly describe an object property using words like "hot", "far", "tall", or "cheap". Although intuitive, these concepts bear a high degree of imprecision and uncertainty, and cannot be modeled using the traditional set theory. Nevertheless, the problem can be tackled successfully by using fuzzy logic.

Basics of fuzzy logic. Unlike conventional logic, in fuzzy logic [9] a predicate may assume any value in a continuous range, usually defined between 0 (totally false) and 1 (totally true). From a set theoretical standpoint, this means that each logic element belongs to a particular set with a certain degree of membership. The function that defines the mapping between the elements of a particular universe of discourse and their degree of membership to a given set is called *membership function*.

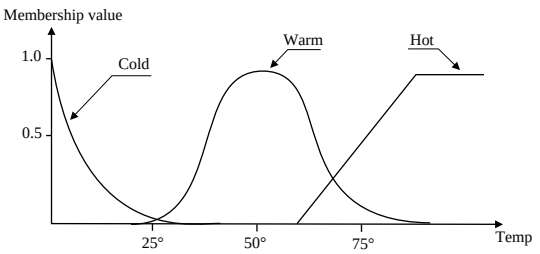

Figure 3: Membership functions and fuzzy sets.

For example, let us consider the problem of characterizing water temperature. When water is freezing at 0°C everybody agrees that it is definitely *cold*—and similarly *hot* when boiling at 100°C. But what about water at 75°C? Modeling this situation entails defining the fuzzy sets, i.e., the intuitive concepts used in the logic descriptions—e.g., *hot*, *warm*, and *cold* in our case. Moreover, each set must be associated to a membership function. Figure 3 shows a possible choice for our example where the value 75°C (that is called *crisp*) belongs to two different fuzzy sets or, in other words, "water at 75°C" is at the same time *warm* and *hot*, with two different degrees of membership.

To enable reasoning, fuzzy logic also provides operators to combine fuzzy predicates in more complex formulas. These are adaptations of the well-known intersection (AND), union (OR), and complement (NOT), to deal with degrees of truth expressed as real numbers. More details can be found in [9].

In LIGHTS, the tuple space contains crisp values, which applications can query using conventional matching or the fuzzy matching provided by `lights.extensions.fuzzy`.

Programming model. In our API, fuzzy sets and their membership functions are combined in what we call a *fuzzy term*. A collection of fuzzy terms represents, in programming terms, a *fuzzy type*. As the reader may argue, matching based on fuzzy logic requires the fuzzy type of two fields to match.

The following code snippet shows how to model the water temperature example with our API:

```
FuzzyTerm ft =
  new FuzzyTerm("warm", new PiFunction(50.0f,25.0f);
FloatFuzzyType temp =
  new FloatFuzzyType("Temperature",-100,100).addTerm(ft);
```

The first line defines a new fuzzy term representing the *warm* concept. A term is defined by a name and a membership function, in this case a `PiFunction` centered at 50°C and with a width of 25°C, yielding the bell shape in Figure 3. Our library provides several pre-canned functions (e.g., `Triangle`, `Trapezoid`, `Ramp`, `Step`, ...), and enables the programmer to easily create her own, by implementing the interface `IMembershipFunction`.

The second line creates a new fuzzy type, and binds to it the previously created term. (Details representing *hot* and *cold* are omitted.) A fuzzy type is characterized by a name and two parameters delimiting its domain. In general, the crisp values in a fuzzy type could be of any nature, and therefore a `FuzzyType` class is provided whose elements can be any `Object` instance. In practice, however, real numbers are used. Therefore, we provide a subclass `FloatFuzzyType`, used in the example.

Integrating fuzzy logic and tuple spaces. We are now ready to describe how to exploit fuzzy matching in LIGHTS. The full API provided by our extension is illustrated by the UML diagram in Figure 4. Two new classes are provided, `FuzzyField` and `FuzzyTuple`, which implement respectively `IField` and `ITuple` and enable use of fuzzy logic at two different levels.

A `FuzzyField` can be included in a conventional template, e.g., a `lights.Tuple` object. In this case, the overridden method `matches` performs matching based on fuzzy logic,

and returns true only if the membership value of the crisp data found in the field being compared is higher than a given threshold, associated to the membership function.A `FuzzyField` is still characterized by type and value, although these are expressed in a fuzzy fashion. In the following code snippet

```
FuzzyField ff = new FuzzyField()
                .setType(Float)
                .setFuzzyType(temp)
                .setFuzzyValue(new FuzzyValue("warm"))
```

a `FuzzyField` is created. First, the type of the crisp values is set, to enable "pre-filtering" of matching values—the basic type matching requirement of Linda is still in place. Then, the fuzzy type defined above for temperature is associated to the field, followed by the "warm" concept. Fuzzy concepts are represented by an instance of the class `FuzzyValue`, which enables the programmer to specify a fuzzy concept. In addition, `FuzzyValue` provides the machinery to specify concepts like "very hot" or "somewhat cold" and automatically adjust the corresponding threshold. Space limitations prevent us from going into further details: anyway, this is performed using well-known techniques [9].

The true power of fuzzy logic, however, is unleashed only when `FuzzyField`s are used in a `FuzzyTuple`. As usual, a `FuzzyTuple` matches another tuple only if all the fields match in order. However, in this case the conjunction of the result of pairwise field matching is not performed using the boolean operator AND, but with its fuzzy counterpart. The method `FuzzyTuple.matches` does not rely on `FuzzyField.matches`, as the latter implements `ITuple.matches` and therefore returns a boolean. Instead, it relies on `FuzzyField.fuzzyMatches`, which returns a float representing the degree of membership of the crisp value in the fuzzy set specified by `FuzzyValue`. If the tuple contains also traditional fields, their `matches` method is invoked, and the boolean return value converted to `0.0f` if `false`, or to `1.0f` if `true`. The float values are then combined by `FuzzyTuple.matches` using the default fuzzy AND operator, or a user-defined one. This feature enables the formulation of complex fuzzy queries, possibly mixed with conventional ones, e.g., retrieving the reading from a sensor that is close and that is recording a cold temperature.

Finally, `FuzzyTuple` also provides a simple language that enables one to write more complex and flexible queries using operators other than AND, also provided by our library. This way, it is possible to write the equivalent of logical formulas, as in:

```
(Distance is not Far) or (Price is Cheap)
```

The query string is submitted through `setAdvancedQuery` and executed as part of the matching algorithm in `FuzzyTuple`.

4.3 Aggregating Data

A very common need in context-aware applications is the ability to deal with aggregated information. For instance, in the experience described in [10], the tuple space holds tuples containing user locations expressed in Cartesian coordinates. The task of selecting the users at a given distance should be ideally as simple as specifying a template with the required distance. In practice, however, it involves computing $\sqrt{(x-x_0)^2+(y-y_0)^2}$, where (x_0, y_0) are the coordinates of the agent issuing the query and (x, y) those of a location tuple. Since Linda semantics does not provide a form of matching based on a function of two or more fields, this matching must be specified entirely outside the tuple space framework, as part of the application logic.

In LighTS, this problem is tackled by introducing the ability to decouple the representation of the tuples stored in the tuple space from those manipulated by the application, using *virtual tuples*.

An example is useful in clarifying their use. Let us consider the possibility of allowing the programmer to "see" the *concrete* tuples stored in the tuple space in the form $p = \langle$?UserID,?int,?int$\rangle$ as if they were instead *virtual* tuples in the form $p' = \langle$?UserID,?int$\rangle$, where the second field of p' is the sum of the last two fields of p. If this were possible, a `rdg(t)` using the virtual tuple $t = \langle$?UserID,50$\rangle$ could match the concrete tuples $\langle$'u15',20,30$\rangle$ and $\langle$'u23',1,49$\rangle$. By substituting sum with distance, we would have found a solution to the aforementioned problem of localizing users. The trick can be accomplished with the following code snippet:

```
ITuple vt = new VirtualTuple(t) {
  public ITuple virtualize(ITuple tuple) {
    ITuple res = new Tuple().add(tuple.get(0));
    IValuedField f = (IValuedField) tuple.get(1);
    int v1 = ((Integer) f.getValue()).intValue();
    f = (IValuedField) tuple.get(2);
    int v2 = ((Integer) f.getValue()).intValue();
    res.add(new Field().setValue(new Integer(v1+v2)));
    return res;
  }
};
vt.add(new Field().setType(UserID.class))
  .add(new Field().setType(Integer.class))
  .add(new Field().setType(Integer.class));
```

The first line creates a new `VirtualTuple` and initializes it with the template used at the application level—the virtual tuple, in our case $t = \langle$?UserID,50$\rangle$. The last three lines define instead the template that filters out the concrete tuples actually present in the tuple space. To enable matching, the concrete tuples must be somehow transformed to fit the format of the virtual tuple. The transformation is specified by the method `virtualize`, which constitutes the remainder of the code snippet where is defined using an anonymous inner class. When a match is requested on `vt`, its overridden `matches` method decides whether the tuple being compared is a match first by comparing it with the standard rules against `vt`'s fields. If this match is successful, the concrete tuple is transformed by calling `virtualize`, and matched against the virtual tuple `t`. This latter matching is governed by the semantics of the `matches` method associated to the dynamic type of `t`, and its result determines the overall matching outcome.

5. IMPLEMENTATION

LighTS is implemented in Java, using the J2SE 1.4 platform. The package `lights.utils` contains additional features that simplify the programmer's life. The interface `Tuplable`, for instance, simplify the task of flattening a structured object into a tuple and vice versa, thus reducing the gap between the object-oriented and tuple space paradigms.

The core `lights` package is only about 150 lines of code. The `adapters` and `extensions` (and especially the `fuzzy` package) bring the total number of lines to 1,500. The sizes of `jar` files are 15Kbytes and 75Kbytes respectively, demonstrating the small footprint of the system.

Without the pretense to be accurate and exhaustive, but with the only intent to get a feel of the performance of

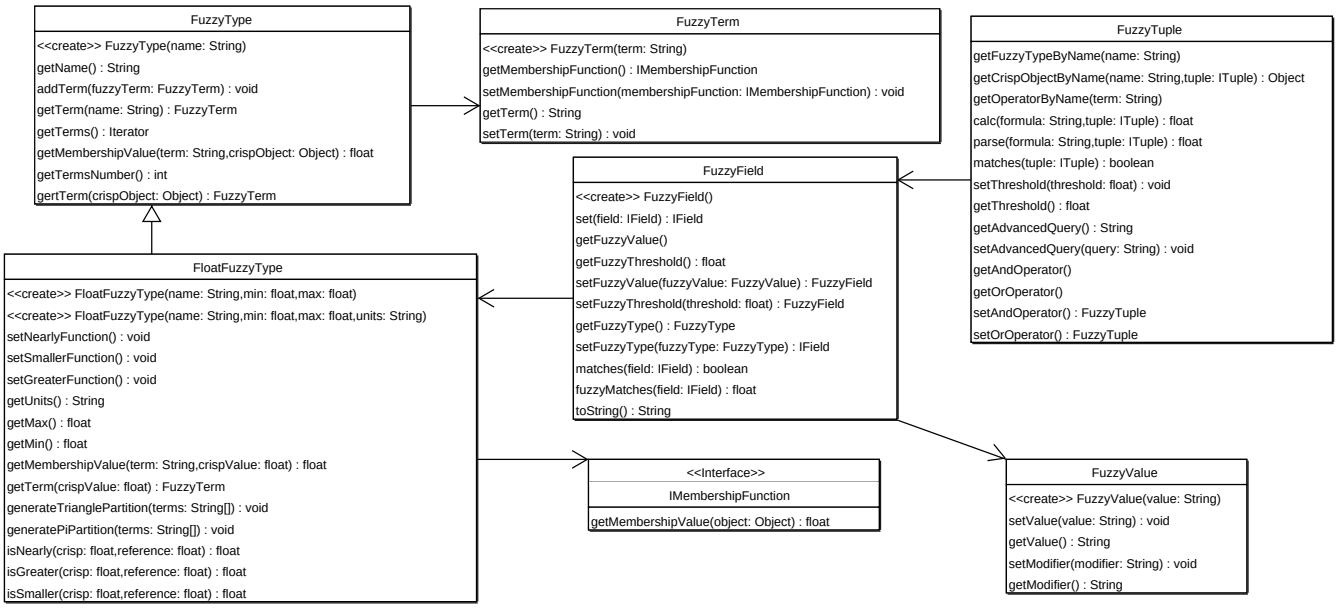

Figure 4: The UML class diagram of the package `lights.extensions.fuzzy`.

LIGHTS, Table 1 reports some tests we ran against commercial systems. These preliminary data show how LIGHTS is always faster than its competitors, which confirms that its lightweight design pays off. As a matter of fact, both the systems compared (probably the most well-known) by design treats local communication as if they were remote, using inter-process communication—a clear loss when only a local tuple space is needed. The one case in Table 1 where LIGHTS is slower than GigaSpaces is probably determined by the techniques exploited in this system to deal with scalability. Definitive results would need to take into account more sophisticated usage profiles—which is nonetheless outside the scope of this paper.

#Tuples	Tuple Size	LighTS	IBM TSpaces	GigaSpaces
100	1000	0.749	0.786	2.536
1000	1000	1.871	4.394	5.534
10000	1000	62.781	120.015	26.611
1000	100	1.806	4.207	5.473
1000	10000	2.111	4.386	5.899
1000	100000	4.166	9.369	10.172

Table 1: A simple performance test on tuple insertion and reading. In each run, we insert several tuples with out, and then read them in sequence with rd. The first field is an integer counter (on which pattern matching is performed), while the second is a byte array. Tests are ran 5 times and results averaged. Tuple sizes are in bytes, times are in seconds. The test machine is a Pentium 4, 2.4 GHz, 1 Gbyte RAM running Sun's JRE 1.4.2 under Debian Linux.

6. CONCLUSIONS

In this paper we presented LIGHTS, a lightweight implementation of the tuple space model. Unlike available systems, LIGHTS is designed with minimality and extensibility in mind. The advantages of this design choice are exemplified by a number of extensions built to support the development of context-aware applications. LIGHTS is released as open source under the LGPL license, and is available at [4].

Acknowledgements. The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, and by the National Research Council (CNR) under the IS-MANET project.

7. REFERENCES

[1] `www.almaden.ibm.com/cs/TSpaces`.
[2] `www.sun.com/software/jini/specs/jini1.2html/js-title.html`.
[3] `www.gigaspaces.com`.
[4] `lights.sourceforge.net`.
[5] `lime.sourceforge.net`.
[6] G. Cabri, L. Leonardi, and F. Zambornelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 2000.
[7] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus-Linda. In *Object-Based Models and Languages for Concurrent Systems*, LNCS 924. Springer, 1995.
[8] D. Gelernter. Generative Communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.
[9] G. J. Klir, B. Yuan, and U. H. S. Clair. *Fuzzy set theory: foundations and applications*. Prentice Hall, 1997.
[10] A. Murphy and G. Picco. Using Coordination Middleware for Location-Aware Computing: A LIME Case Study. In R. D. Nicola, G. Ferrari, and G. Meredith, editors, *Proc. of the 6th Int. Conf. on Coordination Models and Languages (COORD04)*, LNCS 2949, pages 263–278, Pisa (Italy), Feb. 2004. Springer.
[11] A. Murphy, G. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In

F. Golshani, P. Dasgupta, and W. Zhao, editors, *Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 524–533, May 2001.

[12] R. D. Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.

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

[14] A. Rowstron. WCL: A coordination language for geographically distributed agents. *World Wide Web Journal*, 1(3):167–179, 1998.