The LighTS Tuple Space Frawework

and its Customization for Context-Aware Applications

Davide Balzarotti, Paolo Costa, and Gian Pietro Picco

Dip. di Elettronica e Informazione, Politecnico di Milano, Italy

{balzarot,costa,picco}@elet.polimi.it

Abstract

Our experience with using the tuple space abstraction in context-aware applications, evidenced that the traditional Linda matching semantics based on value equality are not appropriate for this domain, where queries often require the ability to match on value ranges, deal with uncertainty, and perform data aggregation. Originally developed as the core tuple space layer for the LIME middleware, LIGHTS provides a flexible framework that makes it easy to extend the tuple space in many ways, including changing the back-end implementation, redefining the matching semantics, and providing new constructs. In this paper, we describe the design and programming interface of LIGHTS, and show how its flexible architecture can be easily extended to define novel constructs supporting the development of context-aware applications.

1 Introduction

The tuple space model, originally introduced by Linda [9] and once popular in parallel programming, is now experiencing a second wave of popularity in the context of distributed and multi-agent systems. Commercial systems (e.g., TSpaces [1], JavaSpaces [2], GigaSpaces [3]) as well as academic ones (e.g., MARS [6], TuCSoN [14], Klaim [13], LIME [12]) are currently available.

In this paper[1], we present LIGHTS, a new, lightweight, customizable tuple space framework. Differently from many of the above systems, LIGHTS was designed with minimality and extensibility in mind, by focusing on providing support for the basic Linda operations in a *local* implementation of a tuple space, which can be used directly (e.g., for

[1] A preliminary version of this paper appeared in [15].

supporting coordination among co-located agents) or as a stepping stone for a more sophisticated distributed tuple space implementation. Indeed, LIGHTS was originally developed by the last author as the core local tuple space support for the LIME system, which builds distributed federation of tuple spaces as well as reactive and transactional features on top of LIGHTS.

In LIGHTS, the lack of distribution and other sophisticated features is compensated by a design that fosters high degrees of customization and flexibility. 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 core, 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.

In particular, in this paper we extend LIGHTS with a number of constructs that are expressly conceived to support context-aware applications. As we discuss in more detail later, tuple spaces can be exploited to store not only application data, but also contextual data like location, data collected by sensors, and other data acquired from the physical environment. This choice empowers application programmers with the ability to deal with both kinds of data—application and context—under a single, unified paradigm, therefore leveraging off the advantages of the tuple space model, e.g., in terms of decoupling. Nevertheless, context-aware applications demand matching rules and tuple space access capabilities normally not found in available tuple space systems, like the ability to match on ranges of values, express uncertainty about data, and perform data aggregation. In this work, we show how these capabilities can be easily built as extensions to our tuple space framework.

Therefore, in this work we put forth two contributions. First, we present the overall architecture and programming interface of LIGHTS and describe its mechanisms supporting customization and extension. Then, we exploit these mechanisms to design and implement extensions geared towards context-aware applications. The latter not only demonstrates the versatility of our framework, but also provides expressive and efficient constructs delivering the power of the tuple space abstraction in this novel application domain, to a level currently not found in available tuple space platforms.

The paper is organized as follows. Section 2 is a concise overview of Linda. Section 3 presents the application programming interface and overall design of LIGHTS, illustrating how the framework can be easily extended both in terms

of performance and semantics. Section 4 discusses the extensions we developed to address some of 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, 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 $\mathbf{out}(t)$ operation, and can be removed by executing $\mathbf{in}(p)$. 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 $\mathbf{rd}(p)$ 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., [16]) provide also *bulk operations*, which can be used to retrieve all matching tuples in one step[2].

3 LighTS: A Lightweight, Customizable Tuple Space Framework

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 extensions features useful for context-aware applications.

[2]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.

3.1 Programming Tuple Space Interactions in LighTS

The core of LIGHTS is constituted by two packages. The `lights.interfaces` package contains the interfaces that model the fundamental concepts of Linda (i.e., tuple spaces, tuples, and fields). Instead, the `lights` package contains a built-in implementation of these interfaces, providing the base for extending the framework.

Tuple spaces Figure 1 shows[3] 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 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

[3]Exceptions are omitted for the sake of readability.

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 ith template field matches the ith 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 relies on the `equals` method—as usual in Java.

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("Authors");
```

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();
t1.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 Additional Programming Features

The package `lights.utils` contains a couple of programming features that, albeit not fundamental, greatly simplify the programming chore.

Accessing Fields by Name Tuples often consist of several fields, to enable a highly selective pattern matching. However, access to these fields is based on their position in the tuple, which makes programming cumbersome since the binding between the field and its meaning remains implicitly encoded in the field position.

To simplify the programming task of accessing a tuple's fields, the package `lights.utils.labels` provides support for associating a symbolic name to a field. Two interfaces, `ILabeledField` and `ILabeledTuple` essentially provide accessors for setting and retrieving a field's label, and for retrieving a field given its label, respectively. In the same package, `LabeledField` and `LabeledTuple` provide specializations of the core classes in the `lights` package, providing support for labels.

To get a feel of why this feature is useful in practice, let us consider a tuple $\langle \textsf{lastName}, \textsf{firstName}, \textsf{phone}, \textsf{dept}, \textsf{salary} \rangle$ representing an employee's data. Printing the full name of the employee followed by her salary is normally achieved by:

```
System.out.println(t.get(0)+" "+t.get(1)+", "+t.get(4));
```

Assuming the proper labels have been attached to the tuple fields, the code above can be rewritten into the more understandable:

```
System.out.println(t.get("Last name")+" "+t.get("First name")+", "+t.get("Salary");
```

As we discuss later on, this simple functionality becomes key in providing enhanced expressiveness in some of advanced features discussed in the rest of this paper.

From Objects to Tuples—and Back Using the tuple space abstraction in the context of an a object-oriented language like Java often forces the programmer to face clashing programming needs. According to the base principles of object-orientation, objects must encapsulate their own data to prevent unauthorized access and to avoid undesired side effects. On the other hand, tuples must expose all of their fields to allow pattern matching. Consider an instance of a class `Employee` holding information about the name of an employee, her phone number, the department she is with, and her monthly salary. If this information is to be stored in a tuple space, there are usually two alternatives. The first one is to represent it as a tuple with a single field containing the `Employee` object. However, this solution greatly limits the

power of pattern matching, preventing queries as "find all the employers working in R&D earning less than $1000 a month". The alternative is for the programmer to manually "flatten" the object into a tuple (e.g., with the same format we used when discussing labels) each time an `out` is performed, and perform the opposite process each time a tuple is retrieved from the tuple space, which is clearly undesirable and awkward. To help the programmer deal with this frequent and error-prone task in a more organized way, LIGHTS provides support through the interface `Tuplable` and the class `ObjectTuple`, shown in Figure 2.

To enable flattening of an object into a tuple, the object must implement the interface `Tuplable`. The method `toTuple` contains the application-dependent code responsible for flattening the object. Transforming the object into a tuple and inserting it in the tuple space can then be done straightforwardly as in

```
ts.out(e.toTuple());
```

where we assume, for instance, that `e` is of type `Employee` and implements `Tuplable` by providing the appropriate code.

Retrieving an object back from a tuple is only a little more complicated. The necessary processing must be encoded by the programmer into the method `setFromTuple`, which allows to set an object's attributes based on the content of the parameter tuple. However, this requires the programmer to manually create an "empty" copy of the object and set its attributes, each time. To further simplify the process, LIGHTS provides the `ObjectTuple` class, which extends the default `Tuple` implementation by providing the ability to "remember" the type of the object a tuple was originally created from. With this facility, a tuple can be read from the tuple space and the corresponding object recreated as in

```
ObjectTuple ot = (ObjectTuple) ts.rd(template);
if (ot.getClassName.equals("Employee")) e = (Employee) ot.getObject();
```

The implementation of `getObject` first invokes the default constructor of the object[4], and then automatically calls `setFromTuple` on the newly created instance by passing `this` as the tuple parameter from which to reconstruct the object state.

Clearly, the solution we just described still requires the programmer to write the code determining how the flattening process is performed, since this is application-dependent. However, it greatly improves the quality and readability of the resulting code, by properly encapsulating this code into the definition of the object, rather than dispersing it into the application code.

[4] Implementation of the default constructor is a requirement for using `ObjectTuple`.

3.3 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, which are exploited in the extension packages included in the LIGHTS distribution.

3.3.1 Changing the tuple space engine

The tuple space implementation in the `lights` core package is very simple[5]. 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 simplify this development strategy, `lights.adapters` provides the building blocks necessary to replace the built-in implementation in `lights`. 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[6]. 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. 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, like those often found in context-aware applications.

[5] Space limitations force us to redirect the reader looking for more details to the online documentation and source [4].

[6] Extensions are not supported by adapters.

3.3.2 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. Here, we briefly describe some of these extensions.

Field matching As an example of how to redefine matching between fields, the class `SubtypeField`, takes subtype compatibility into account. 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`. Analogously, `NotEqualField` supports matching on inequality. 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 libraries.

Tuple matching Pattern matching between tuples can be redefined similarly. `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`.

Another useful feature, with a more complex implementation, is provided by `BooleanTuple`, which extends `Tuple` to enable pattern matching by using arbitrary boolean expressions over a tuple's fields—not just their *AND* conjunction as in standard matching. Let us consider an application where printers are modeled as a tuple ⟨kind, numJobs, ppm⟩, and the programmer needs to find a laser printer such that either its spool queue is empty or it can print at 15ppm. The programmer can obviously compensate for the absence of an *OR* operator by performing multiple queries in sequence—in our case, one for a laser printer with an empty queue and, if this fails, one for a laser printer at 15ppm. However, besides forcing the programmer to use a more verbose programming idiom, this solution is rather inefficient, since it requires multiple traversals of the tuple space. The problem is exacerbated in the case of the bulk operations `rdg` and `ing`, where *all* the multiple queries must be executed.

Instead, `BooleanTuple` behaves as a traditional tuple but in addition it provides a `setMatchingExpression` method that allows users to specify a template in the form of a logical formula. A valid formula must be a well-formed boolean expression, which can contain any combination of *AND*, *OR*, and *NOT* operators, specified using the Java syntax. The fields involved in the expression can be specified either by using their position in the tuple, prepended by the # character, or, if they are instances of `LabeledField`, by using directly their label.

For instance, with reference to the previous example, let us assume that two actual fields labeled `laser` and `empty` are available respectively to match a laser printer and one with a queue with zero jobs, and that a third field takes care of matching a value of 15. With these definitions, which we omit due to space limitations, we can easily search for the desired printer:

```
BooleanTuple template = new BooleanTuple().add(f1).add(f2).add(f3);
template.setMatchingExpression("laser && (empty || #3)");
ITuple tuple = ts.rdp(template);
```

It is worth noting how the advantages brought by the customization of the matching semantics we described thus far are not limited to the tuple space access using the traditional Linda operations, but may extend to other constructs provided by alternative models. For instance, several systems offer reactive features, either event-based as in TSpaces, JavaSpaces, MARS [6], and TuCSoN [14], or state-based as in LIME [12]. In all these systems, some kind of reaction whose behavior is specified by the programmer, is triggered when a tuple matching a given template is manipulated through an operation or observed in the tuple space. Redefining the template used in these operations may greatly enhance their expressive power.

At the same time, special care is needed if these customized features are being used in a distributed setting. Consider even a simple client-server scheme, where a tuple space is being accessed remotely by several clients. If the client uses an extension (e.g., `RangeField` or a programmer-defined one) in a query operation, the corresponding code must be somehow present on the server host for the tuple space to be able to apply the desired matching. Pre-deploying the classes is possible only under the assumption of a closed system: if programmers are able to define their own extensions, appropriate mechanisms (e.g., involving code mobility [8]) must be in place.

Finally, many other extensions are clearly possible. One could easily implement SQL- or XML-based matching, and many others. Thus far, the development of extensions has been driven by pragmatic needs that arose in our experiences when using LIGHTS in combination with the LIME middleware. In the next section, we illustrate some extensions we found useful in developing context-aware applications.

4 Supporting Context-Aware Applications

As we discussed in the introduction, 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. Indeed, the motivation for the work described in this paper came from an experience in building a simple location-aware application in LIME [11], in which we realized precisely the aforementioned shortcomings of the traditional tuple space model. Here we describe briefly the outcomes of this experience, in that they provide the rationale for the features we describe in this section.

The work in [11] describes 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 implementation exploits tuple spaces as repositories for context information—i.e., location data in this case. The LIME primitives are used to seamlessly perform queries not only on a local tuple space, but on all the spaces in range. For instance, a user's location can be determined by performing a read operation for the location tuple associated to the given user identifier. The "lesson learned" distilled from this experience 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. Interestingly, in this experience the idea was demonstrated using the distributed tuple space implementation provided by LIME, but the conclusion we just made fully holds also in the case where an entirely local tuple space is used to coordinate the activities of co-located multi-agents.

Nevertheless, as discussed in [11], 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 [11] 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". Moreover, context-aware applications frequently pose another requirement, namely, the need for *aggregation*. Data comes from multiple sources, with multiple formats, and at different levels of abstraction. On one hand, it is useful to store the raw data in the tuple space, to provide applications or agents to process it directly. However, in other situations, it is desirable to access the data through some higher-level view, where the values contributed by multiple tuples are accessed as a single, aggregated value (e.g., the average), or where tuples with a given format (e.g., holding a location's coordinates) are interpreted in a different way (e.g., distance from a given point).

These needs sometimes surface also in conventional applications, but they are definitely exacerbated and more fundamental in context-aware ones. In the rest of this section we present our extensions to LIGHTS fulfilling these requirements and therefore supporting the development of context-aware applications.

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). Building this capability on top of a conventional system that provides only exact value matching entails considerable programming effort and computational overhead. 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 methods for specifying the lower and upper bounds of the value range and whether they are included in it, as shown in Figure 3. 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 *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 [10] 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*.

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 4 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 [10].

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 4. 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 most of the times. 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 5. 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` evaluates 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 that can be chosen by the user. 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 membership function. Space limitations prevent us from going into further details: anyway, this is performed using well-known techniques [10]. Figure 6 illustrate pictorially the difference between traditional matching and matching with fuzzy values.

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 this implements `ITuple.matches` and therefore returns a boolean. Instead, it relies on the method `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 is recording a cold temperature.

Class	Field Composition
`Tuple`	Logical *AND*
`BooleanTuple` (default)	Logical *AND*
`BooleanTuple` (using `setMatchingExpression`)	Any expression containing boolean operators
`FuzzyTuple` (default)	Fuzzy *AND*
`FuzzyTuple` (using `setMatchingExpression`)	Any expression containing boolean and fuzzy operators

Table 1: Comparing the expressive power of various matching semantics.

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) || (Price is Cheap)
```

The reader has probably noticed the analogy with the `BooleanTuple` class we described in Section 3.3.2. Indeed, `FuzzyTuple` extends `BooleanTuple`, and overrides the method `setMatchingExpression` to enable the definition of arbitrary predicates. Some important differences must be underlined, however. First, the introduction of the `is` operator that returns the degree of membership to the given `FuzzyTerm`. Second, `FuzzyTuple` supports not only arbitrary boolean expressions, but also expressions that involve user-defined fuzzy operators, as we discussed above. Third, the values that are involved in the expressions (e.g., `Far` and `Cheap`) can be fuzzy values. Table 1 provides a concise albeit informal comparison of the expressive power provided by conventional tuples, `BooleanTuple`s, and `FuzzyTuple`s.

4.3 Aggregating Data

A need often arising in context-aware applications is the one for the ability to deal with aggregated information. This is addressed in LIGHTS by two distinct mechanisms, one enabling aggregation over fields in the context of a given tuple, and the other enabling aggregation over multiple tuples contained in the tuple space. In both cases, the application programmer is provided with a way to specify the portion of data to be aggregated and the semantics of the data transformation involved. Both mechanisms are described next.

4.3.1 Aggregating Fields: Virtual Tuples

A concise example helps in defining the need we address. Let us consider a tuple space containing location information, where each tuple holds the location of a user expressed in Cartesian coordinates. (This solution was actually used in the experience described in [11].) 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 the 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.

LIGHTS tackles the problem by decoupling the representation of the tuples stored in the tuple space from those manipulated by the application, by means of *virtual tuples*. Again, 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. Figure 7 illustrates the concept graphically.

Using our LIGHTS extension, this feature can be provided by 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 in the example code above 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 by first 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.

4.3.2 Aggregating Tuples: Tuple Space Views

Tuple virtualizers provide an elegant way to customize on the fly the tuple representation, therefore enabling also aggregation over the tuple fields. Nevertheless, their scope is limited to a single tuple, whereas context-aware applications often demand aggregation over multiple tuples. A typical example is provided by environmental monitoring, where the data independently collected by multiple sensors is often averaged before being provided to applications, to make the data more resilient to transient variations. Imagine an application that receives tuple containing data sensed by multiple sensors (e.g., for temperature, light, acoustic phenomena) and stores them in the tuple space. Data is sometimes accessed in its raw form, and sometimes in aggregate form (e.g., through its average). It would be useful if the programmer were able to see this second option again as a tuple space, without the need to manually compute over and over the aggregation by herself.

This problem is dealt with in LIGHTS by introducing the notion of a *tuple space view* defined over a tuple space. The tuples contained in a tuple space view are obtained from a subset of those in the original tuple space, through an automatic, application-defined transformation. In a sense, a view realizes on the tuple space what a virtual tuple does on a single tuple, by providing a *virtual tuple space* built on top of the concrete tuple space associated to it.

A view is created by simply invoking its constructor that, as shown in Figure 8, expects as parameters the tuple space the view is built upon as well as the rules to maintain it. Once the view is created, as shown in the figure only the `rdp` and `rdg` operations are available, since it is not possible to manipulate directly the view.

The transformation from the concrete tuples in the tuple space to those in the view is encapsulated in the set of `Aggregator` objects passed as a parameter to the constructor. Each of these object effectively defines a function from

a set of tuples in the concrete tuple space (specified by the template) to another set of tuples to become part of the tuple space view. The abstract class `Aggregator`, also shown in Figure 8, provides accessors for the template, as well as an abstract method `aggregate` that is expected to embody the aforementioned programmer-defined function. When an operation is invoked on the `TupleSpaceView`, this method is automatically called and supplied with the set of concrete tuples matching the template. In turn, `aggregate` returns the virtual tuples logically belonging to the `TupleSpaceView`. The `Aggregator` class contains a built-in template matching all tuples in the tuple space, defined using `PrefixTuple`. Therefore, if no template is set, the `aggregate` method operates on all the tuples in the target tuple space. Figure 9 illustrates the concept.

Note how the behavior of the `aggregate` method does not necessarily entail collapsing multiple tuples into one or more. For instance, in some cases it maybe useful to "join" multiple tuples containing values about different physical entities into one or, in turn, "split" long tuples into their individual values. For instance, with reference to the aforementioned environmental monitoring application, imagine that the application needs to determine whether there are people in a given area, based on whether there is a high temperature and a high noise. Using the conventional features, the application should retrieve all the tuples with high temperature and all those with high noise, and then manually check whether one or more rooms exist that belong both tuple sets. Using views, the programmer can specify how to combine temperature and light readings from the same room in a single tuple, and then query the view as desired, e.g., using a fuzzy template ⟨Room=*, temperature=high, noise=high⟩. Therefore, ultimately, the nature of the transformation performed by `aggregate` is entirely up to the programmer.

In our current implementation, the `aggregate` methods are called every time a read operation is invoked, therefore re-building the view dynamically each time. This straightforward solution guarantees that the view is always consistent with the tuple space it builds upon, but it may generate a performance problem in the case the tuple space contains a large number of tuples and the view operations are invoked frequently. An alternative strategy is to cache the result of previous executions of the `aggregate` method. This solution avoids unnecessary computation if the associated tuple space has not changed, but it requires a tighter integration between the `TupleSpaceView` class and the tuple space holding the concrete tuples, since the latter must somehow notify the former when a tuple of relevance for the view has been inserted or removed, and therefore the view must be recomputed. This latter design can be easily accommodated by constraining `TupleSpaceView` to operate in conjunction with a subclass of `TupleSpace` (or any other class implementing `ITupleSpace`) providing the necessary coupling. We are currently investigating more optimized solutions to based on this ideas.

As an example of how to program and exploit tuple space views, consider a context-aware application monitoring a physical environment containing several sensors. Each sensor records the temperature and inserts it in the tuple space together with its location, using a tuple ⟨x, y, temp⟩. Suppose we are interested in retrieving the average value in the square zone defined by (x_{min}, y_{min}) and (x_{max}, y_{max}), e.g., because a fire is reported in that area and finer-grained monitoring is necessary. First, we need to define the aggregation function computing the average temperature. This is accomplished by extending the `Aggregator` class and implementing the `aggregate` method:

```
class AvgAggregator extends Aggregator {
  public ITuple[] aggregate(ITuple[] tuples) {
    float res = 0;
    for (int i=0; i<tuples.size; i++)
      res = res + tuples[i].get("temp").getValue();
    res = res/tuples.size;
    return new Tuple().add(new Field().setValue(new Float(res)));
  }
}
```

Next, we create a template using the `RangeField` class we introduced in Section 4.1 to select only those tuples whose location belongs to the desired zone, and instantiate our aggregator by restricting its operation to these tuples, by passing it the template just defined.

```
RangeField xf = new RangeField().setLowerBound(new Float(xmin),true)
                                   .setUpperBound(new Float(xmax), true);
RangeField yf = new RangeField().setLowerBound(new Float(ymin),true)
                                   .setUpperBound(new Float(ymax), true);
ITuple template = new Tuple().add(xf).add(yf).add(new Field().setType(Float.class));
Aggregator a = new AvgAggregator().setTemplate(template);
```

Now, we are ready to generate the view by passing the tuple space `ts` it operates upon and our aggregator:

```
TupleSpaceView view = new TupleSpaceView(ts,{a});
```

We can now read from the tuple space view as if it were a normal tuple space, containing ⟨Float⟩ tuples, as defined by our aggregator:

```
ITuple avgTemplate = new Tuple().add(new Field().setType(Float.class));
ITuple t = view.rdp(avgTemplate);
```

#tuples	tuple size	LighTS	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 2: 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.

5 Implementation

LIGHTS is implemented in Java, using J2SE 1.4. 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 LIGHTS, Table 2 reports some tests we ran against some well-known commercial systems. These preliminary data show how LIGHTS is always faster than its competitors, which confirms that its lightweight design pays off. In part, this can be attributed to the fact that the systems considered do not distinguish between local and remote communication, always using inter-process communication—a clear loss when only a local tuple space is needed. The one case in Table 2 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.

6 Conclusion

In this paper we presented LIGHTS, a lightweight, customizable framework supporting the tuple space abstraction made popular by Linda, and exploited its flexible architecture to provide dedicated constructs for the development of

context-aware applications. We illustrated the architecture and application programming interface of LIGHTS, motivated the use of tuple spaces for context-aware applications and the related challenges, and showed how novel support for this domain can be easily built on top of LIGHTS. Future work will address additional optimizations of the mechanisms we described here, and integration within the LIME middleware.

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.

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. Zambonelli. 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] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, May 1998.

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

[10] G.J. Klir, B. Yuan, and U.H. Saint Clair. *Fuzzy set theory: foundations and applications*. Prentice Hall, 1997.

[11] A.L. Murphy and G.P. Picco. Using Coordination Middleware for Location-Aware Computing: A LIME Case Study. In *Proc. of the 6th Int. Conf. on Coordination Models and Languages (COORD04)*, LNCS 2949, pages 263–278. Springer, February 2004.

[12] 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)*, pages 524–533, May 2001.

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

[14] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In *Proc. of the Symp. on Applied Computing (SAC'99)*, February 1999.

[15] G.P. Picco, D. Balzarotti, and P. Costa. LIGHTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications. In *Proc. of the 20th ACM Symp. on Applied Computing (SAC05)*, Santa Fe (New Mexico, USA), March 2005. ACM Press.

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

```
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.

```
public interface Tuplable {
  ITuple toTuple();
  void setFromTuple(ITuple tuple);
}

public class ObjectTuple extends Tuple {
  public ObjectTuple(Class c);
  public Tuplable getObject();
  public String getClassName();
}
```

Figure 2: Types for flattening objects into tuples—and vice versa.

```
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 3: The class `RangeField`.

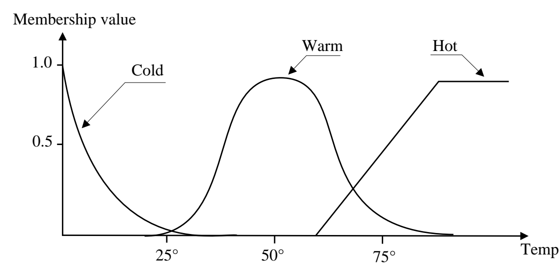

Figure 4: Membership functions and fuzzy sets.

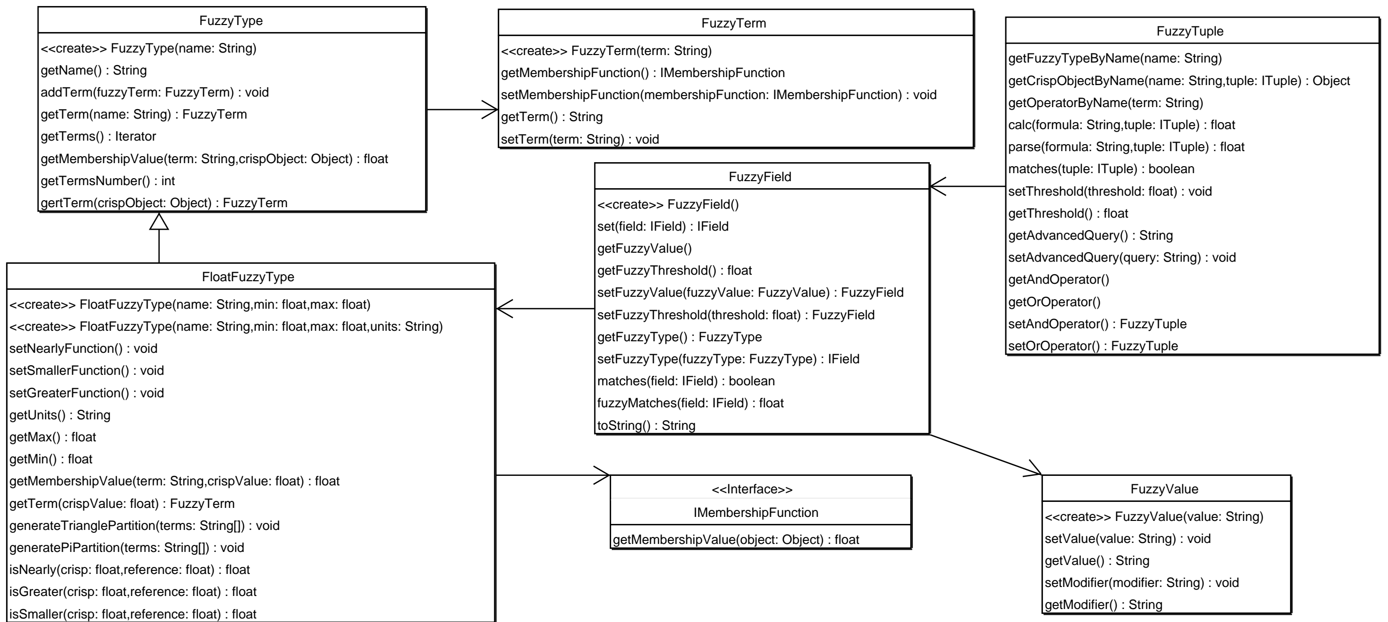

Figure 5: The UML class diagram of the package `lights.extensions.fuzzy`.

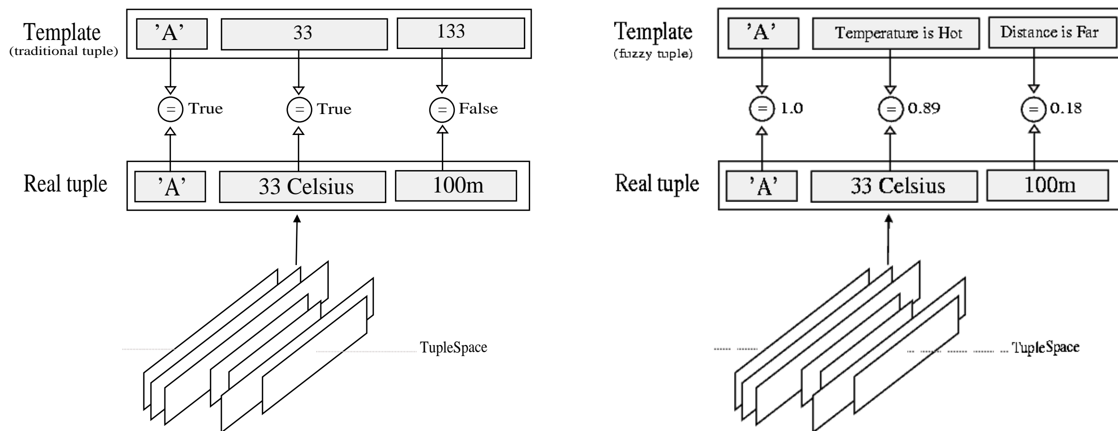

Figure 6: Replacing traditional matching (left) with matching based on fuzzy values (right).

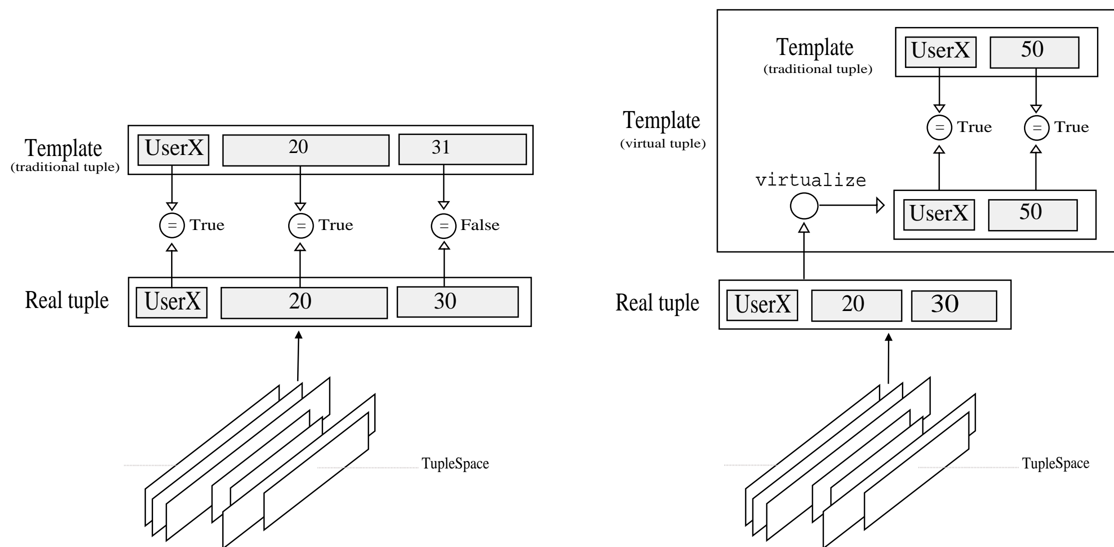

Figure 7: Replacing traditional matching (left) with matching against a programmer-defined *virtual tuple* (right).

```
public class TupleSpaceView {
  public TupleSpaceView(ITupleSpace ts, IAggregator[] a);
  public ITuple rdp(ITuple template);
  public ITuple[] rdg(ITuple template);
}

public abstract class Aggregator {
  public void setTemplate(ITuple template);
  public ITuple getTemplate();
  abstract ITuple[] aggregate(ITuple[] tuples);
}
```

Figure 8: Dealing with tuple space views: `TupleSpaceView` and `Aggregator`.

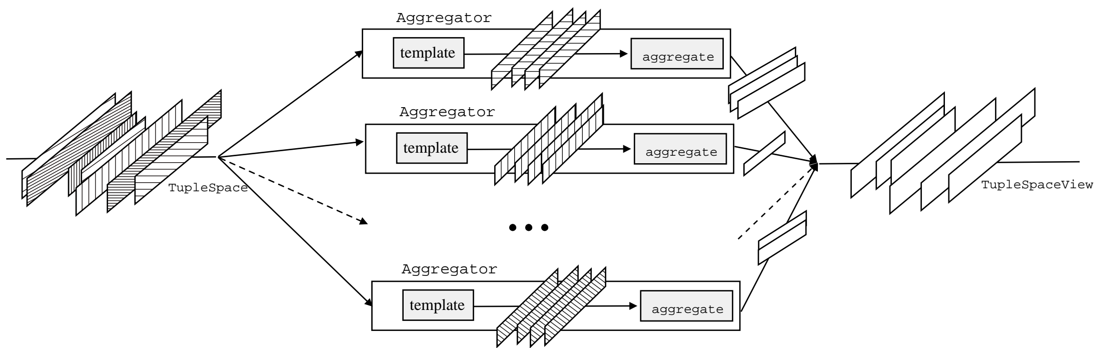

Figure 9: Aggregating multiple tuples using a tuple space view.