Geo-Linda: a Geometry Aware Distributed Tuple Space

Julien Pauty
K.U. Leuven

Paul Couderc
INRIA

Michel Banâtre
INRIA

Yolande Berbers
K.U. Leuven

K.U. Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
firstname.lastname@cs.kuleuven.be

INRIA
Campus de Beaulieu
35042 Rennes, France
firstname.lastname@irisa.fr

Abstract

This paper presents Geo-Linda, a physically distributed tuple space. Geo-Linda targets ubiquitous computing applications involving the detection of movement patterns of objects and people, such as the meeting of two people or the insertion of a product in a shopping cart, the loading of container in a boat... Existing distributed tuple spaces suffer two main limitations: (1) they cannot detect precisely movement patterns, leading to inconsistencies between the state of applications and the physical world; (2) they cannot detect several kinds of movement patterns. Geo-Linda tackles these two limitations, by introducing a geometric addressing mode and two new reading operations to access the tuple space. In this paper, we illustrate Geo-Linda with several small examples and a full application.

1. Introduction

Ubiquitous computing [20] aims to provide users with a seamless access to computing resources and services. To this end, ubiquitous computing distributes in the physical space computing and sensing capabilities. Ubiquitous computing applications are used from the physical space and linked to humans' daily activities. This link to human activities implies that numerous applications are synchronized on movement patterns of objects and people, such as: the arrival of the user at a given location, the meeting of two users, the insertion of a book in a book shelf, the stopping of a bus... Applications synchronized on movement patterns include: public transport applications [1, 6], electronic supermarkets [17] and hazard detection [18].

We distinguish two approaches to detect movement patterns: the virtual approach and the physical approach. The virtual approach relies on a localization and communication infrastructure. The overall goal is to create a virtual model of the physical world in a service platform and to analyze this model in order to detect movement patterns. The virtual approach implies that mobile peoples and objects regularly update their location in the service platform, in order to keep up the model consistent with the state of the physical world. The virtual approach can raise a scalability problem: the service platform delivers services to all users; if the number of mobile objects and people is too large the service platform becomes a bottleneck.

The physical approach detects movement patterns with coordination protocols between people and objects. To this end, people and objects carry wireless devices and communicate only with devices included in their communication range; devices that are nearby. The rationale of the physical approach lies in the fact that movement patterns involve coordination between neighboring devices and can be detected without a global model of the physical world. For example, a typical coordination protocol monitors the network connectivity to detect meetings of devices: if the communication range of the devices are adapted to the size of their carrying entity, detecting a new device in the communication range consist in detecting a meeting with this device. The physical approach relies on local coordinations and does not suffer from the scalability problem. In the rest of this paper we focus on the physical approach.

Physically distributed tuple spaces have proved to be efficient to implement the physical approach. Such a tuple space embeds a local tuple space on each device. The tuple space viewed by a device A corresponds to the union of the local tuple spaces of the devices included in A's communication range. Movement patterns are reflected in the viewed tuple space by tuple appearances and disappearances: when a device enters in the communication range of A its tuples are inserted in the visible space of A.

Despite their advantages, existing physically distributed tuple spaces face two main limitations: (1) there are several kinds of movement patterns that they cannot detect; (2) they cannot detect precisely movement patterns, leading to inconsistencies between the state of the physical world and the state of the applications. In this paper, we present Geo-Linda. Geo-Linda extends the notion of physically distributed tuple space, in order to detect precisely multiple kinds of movement patterns.

Next section details the notion of physical synchronization. A physical synchronization consists in synchronizing an application on the occurrence of a movement pattern. We will see that physically distributed tuple spaces are especially convenient to program such synchronizations. Section 3 presents Geo-Linda. Geo-Linda relies on a geometric addressing mode to detect precisely movement patterns. In order to detect new kinds of movement patterns, Geo-Linda introduces two new reading operations. Section 4 presents an application programmed with Geo-Linda. Section 5 presents the main implementation aspects and evaluation results. Before concluding, we present in section 6 related works.

2. Physical synchronization

2.1. Physically shared data space

A physically shared data space is distributed on several wireless devices. Each device embeds a short range communication interface which enables it to communicate with other devices included in its communication range. We call connected devices, the devices with which a device can communicate. Each wireless device embeds a local data space and can access to the data stored on devices located in its communication area. Figure 1 shows an example shared data space. We have three devices A, B and C, and each device embeds data. A and C are located inside B's communication area, so the visible data space of B is the union of A, B and C's local space. Note that a shared data space does not rely on an ad-hoc routing protocol: the visible data tuple space of a device is the union of connected devices' local data space.

The content of the visible data space of a device A reflects the arrivals and departures of devices from the A's communication area. When a device B enters in A's communication area, B's data are added to A's visible data space. Conversely, when B leaves the communication are of A, its data are removed from A's visible data space.

Several systems propose such a mechanism of physically shared data space. PERSEND [19] relies on shared databases: each device embeds a local database. Devices access to their visible data space via continuous queries. A continuous query is a standard query which remains active in the database. A traditional continuous query is evaluated

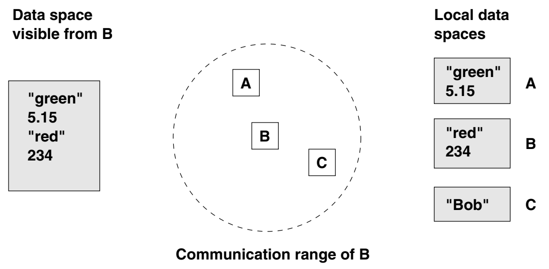

Figure 1. A physically shared data space. Only B's communication area is represented

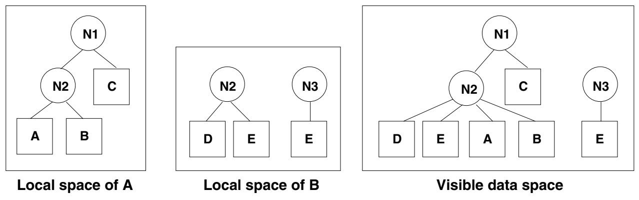

Figure 2. Example of the Peerware sharing mechanism

each time records are inserted or deleted from the database. From the point of view of a device's visible data space, device arrivals correspond to record insertions and device departures correspond to record deletions. LIME [15] and SPREAD [3] are two physically distributed tuple spaces. Each device embeds a local tuple space. The visible data space of a device is the union of its local tuple space and the tuple spaces of the devices located in its communication area. Peerware [4] proposes a sharing mechanism based on a tree structure, composed of named nodes and leaves. Each device stores its data in several local trees and leaves correspond to the data shared by devices. When several devices are connected, nodes that have the same name are represented by the same node in the visible tree (cf. figure 2).

2.2. Physical synchronization

In the preceding section we introduced four systems to share data between connected devices. In the rest of this article we focus on physically distributed tuple spaces, such as SPREAD and LIME.

A device can access the data shared by connected devices with reading operations. Physically distributed tuple spaces propose reading operations similar to the operations proposed by Linda [5]. A reading operation takes the following form: `read<p>`, where `p` is a tuple pattern. A read-

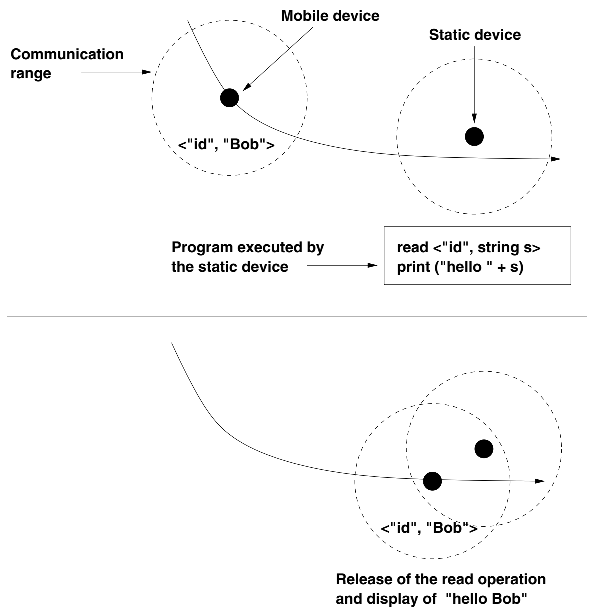

Figure 3. A physical synchronization

ing operation stays blocked until a tuple that matches `p` is available. A tuple is available if it is in the visible tuple space of the device which executes the operation.

The blocking aspect of the reading operation enables us to synchronize applications on meetings of devices. On figure 3, we have a mobile device M and a static device S. The local tuple space of M contains the tuple `<"id", "Bob">`. Initially, S searches for a tuple which must match the pattern `<"id", string s>`. The read operation stays blocked because S's local tuple space contains no matching tuple and there is no connected device. When M enters the communication range of S, M's tuple becomes available to the read operation and the program of S resumes.

With respect to the scale of the entities involved in the movement pattern, the action "entering in the communication range of device A" can actually correspond to "meeting device A. For example, the meeting of two people can be detected with an acceptable accuracy with bluetooth interfaces having a 10 meters range. However, detecting such a meeting with WiFi interfaces is not possible, due to the too large communication range of these interfaces.

The mechanism of physical synchronization fits well ubiquitous computing applications involving movement patterns of people and objects. Indeed, with reading operations, we can easily synchronize code execution on these meetings. Consider the program executed by the static device on figure 3, the statement following the read operation is executed when a matching tuple becomes available, i.e. when the mobile device meets the static device.

2.3 Limitations of the physical synchronization mechanism

The mechanism of physical synchronization suffers from two limitations: (1) we cannot detect precisely the devices' meetings; (2) we can only detect meetings of devices, we cannot detect other kinds movement patterns such as separations of devices.

The first limitation is due to the meeting detection method. As we said the preceding subsection, meetings are detected by monitoring the network connectivity. Therefore, when a read operation is released, we only know that a device is inside the communication range of the device executing the operation; we do not know the relative location of the devices. For example, we cannot distinguish a device arriving on the left from a device arriving on the right. Moreover, if the communication range of devices is too important, the detection of the meeting is not enough precise; we can have the situation where the devices can communicate whereas they are too far to consider that they actually met.

The second limitation is due to the semantic of the read operation. The `read(p)` operation stays blocked until a tuple that matches the pattern `p` is available. This operation detects the "arrival" of a tuples, and so the arrival of the device which carries this tuple. If a tuple is already present the `read` operation cannot detect the arrival of a *new* device. Similarly, the `read` operation cannot detect the departure of a device, which would correspond to the disappearance of a tuple from the tuples space.

In the next section we present Geo-Linda, which is a physically shared tuple space that tackles these two limitations.

3. Geo-Linda

In the preceding section, we have presented the principle of a physically shared tuple space. In such a tuple space, reading operations enable us to synchronize a program executed by a device on the detection of a meeting with another device. We call this synchronization mechanism physical synchronization. A device A detects a meeting with a device B by detecting the arrival of B in its communication area. We have seen that this detection principle is not enough precise, with respect to the communication range of the communication technology used. We have also seen that reading operation enables us detecting only meetings is not sufficient to program several applications.

In the following subsections, we present Geo-Linda, which is a physical share tuple space enabling us to detect precisely different kinds of movement patterns of devices. To this end Geo-Linda relies on geometric addressing mode and proposes new reading operations.

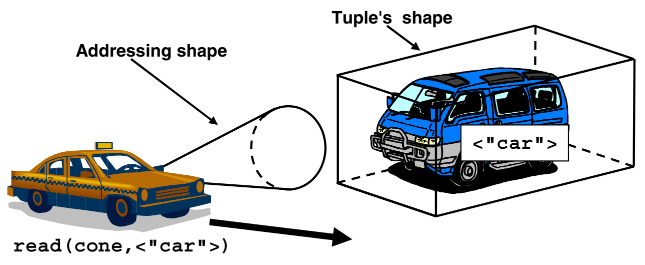

Figure 4. Geometric addressing. The read operation will be released when the cone intersects the box.

3.1. Geometric addressing

In this section we present the geometric addressing mode, which enables us to accurately detect movement patterns. Indeed, in the preceding section, we have seen that we must detect accurately movement patterns, in order to avoid inconsistencies between applications state and the physical space state, such as detecting a meeting between two people which did not yet happen and may never happen.

To detect precisely movement patterns, Geo-Linda associates a volume to each tuple and a volume to each reading operation. We call tuple's shape the volume associated with a tuple. We call addressing shape the volume associated with a reading operation. A reading operation is released when the shape of a matching tuple intersects with the addressing shape of the operation. Tuples' shape and addressing shapes enable us to define the geometric configuration of a movement pattern. For example, suppose that cars signal their presence by publishing a tuple `<"car">`. The shape of this tuple is a box encompassing the car. To detect a car on its left, a car executes an operation such as `read(cone,"car")`, with `cone` being a cone located on the left of the car (see figure 4). When this operation is released, the car has just detected a movement pattern such as: "a car arrived on my left".

Tuple's shape and addressing shapes can be chosen among the following volumes: sphere, cylinder, cone, box, sector and point. Geo-Linda proposes only elementary volumes to keep up the reactivity of reading operations. Indeed, reading operations must perform intersection tests between addressing shapes and tuples' shape, and the duration of these tests directly depends on the complexity of the shapes.

The programmer defines a tuple's shape relatively to the location and the orientation of the device which publishes this tuples. Similarly, he defines the addressing shape of a reading operation relatively to the location and orientation of the device which executes this operation. In this way, like tuples, the defined shapes move with the device. To perform intersection tests, a device needs to know the relative coordinates of the involved shapes. To this end, we equip each device with a location mechanism, such as UbiSense tags [1].

The geometric addressing mode enables the programmer to prevent inconsistencies. Indeed, the programmer should define tuples shapes and addressing shapes so that, when the shapes intersect, the corresponding movement pattern actually happened. In the preceding example, when the cone intersects the box, the `read` operation is released and we detect that a car is actually on the left of car who executes the operation.

3.2. Detecting movement patterns

In order to detect several kinds of movement patterns Geo-Linda proposes several reading operations. Indeed, we have seen in the preceding section that traditional Linda operations do not enable us to detect movement patterns such as separation of device. To address this limitation, Geo-Linda proposes two new reading operations: `lostOne` and `readOnce`.

Geo-Linda proposes several reading and writing operations to access the tuples space. The `out(s,t)` operation enables a device to publish the tuple `t` associated with the shape `s`. The tuple `t` is stored inside the local tuple space. The `drop(t)` operation deletes a *local* tuple matching `t`.

Geo-Linda proposes four reading operations:

- `read(s,p)`, which returns a tuples that matches the pattern `p` and whose shape intersects the addressing shape `s`.
- `take(s,p)`, which returns a tuples that matches the pattern `p` and whose shape intersects the addressing shape `s`. The matching tuple is withdrawn from the tuple space.
- `readOnce(s,p)`, which returns a *new* tuple that matches the pattern `p` and whose shape intersects the addressing shape `s`. This tuple has not been read before with the `readOnce` operation.
- `lostOne(s,p)`, which returns a tuple that matches the pattern `p`. The shape of the returned tuple intersects the addressing shape `s` and has disappeared from the visible tuple space of the entity which executes the operation. The matching tuple must have been read before with the `readOnce` operation.

If several matching tuples are available, the `read` and the `take` operations select one them and return it. We do not specify how this tuple is chosen. If no matching tuple is

[1]`www.ubisense.net`

available, the four reading operations remain blocked until a matching tuple is available.

Geo-Linda generalizes the principle of physical synchronization: each reading operation detects a movement pattern and the program code following this operation is synchronized on the occurrence of the movement pattern.

The operations `read` and `take` are standard Linda operations. The `readOnce` and `lostOne` operations are new operations which enable us to synchronize programs on new movement patterns. We now illustrate how these reading operation can detect movement patterns.

3.2.1 Meeting detection

We have already seen that the `read` operation enables us to detect meetings of two devices. To this end, the first device publishes a tuple with an `out` operation and the shape of this tuple encompasses the entity. The second entity executes a `read` operation. The addressing shape of the `read` operation defines the meeting area and the program code executed after the `read` operation is synchronized on the meeting of these devices. The geometric setting of this meeting is defined by the addressing shape and the tuples shape.

The semantic of the `read` operation is also useful for applications involving a group of entities of the same kind, such as cars. When several matching tuples are available the `read` operation selects one of them and returns it, detecting one entity of the group. Therefore, if every entity carries a tuple following the same pattern `<"car", string model>`, an operation such as `read(add_shape, <"car", string model>)` enables an entity to detect the presence of at least one car in `add_shape`.

3.2.2 Combining movement detection and state change

Like the `read` operation, the `take` operation enables us to detect meetings of two entities, but the tuple involved in the detection can participate only to one detection. Indeed, once the `take` operation is finished, the tuple has been withdrawn from the device that initially carried it.

More generally, the `take` operation enables us to detect movement patterns that involve a change state of one the devices; when the movement pattern is finished the state of a device has changed. For example, consider a pedestrian stopping a taxi. Once the taxi has stopped, the state of the pedestrian has changed: he does not want to take a taxi anymore. Such movement patterns are also present in ticketing applications, once a person has entered a museum or a bus he does not carry his ticket anymore. If we consider a person who takes a plane, she undergoes several state changes: registered, custom checked, boarded. In the tuple space, these state changes are reflected by withdrawals of tuples.

We illustrate the `take` operation by showing how a pedestrian can "stop" a taxi. The taxi executes a `take` operation: `take(cone, "rq_taxi")`. The addressing shape of the operation is a cone, located in front of the taxi. To find a taxi, the pedestrian publishes a tuple `<"rq_taxi">` whose shape is a cylinder encompassing the pedestrian. When the `take` operation is released, a sound alert is broadcast inside the taxi to notify the driver that a nearby pedestrian wants to take his taxi. Contrary to the `read` operation, the `take` enables us to keep the tuple space consistent with the physical space: the `take` operation removes a tuple from the tuple space, in order to reflect the state change of the pedestrian.

3.2.3 Arrival and departures of entities

The `readOnce` and `lostOne` operations are usually used to monitor the arrivals and departures of entities of the same kind from an area of the physical space. We use the `readOnce` operation to detect the arrival of a new entity and the `lostOne` operation to detect the departure of an entity which has been previously detected. The `readOnce` operation enables the applications to detect movement patterns such as: "a new entity has entered into the addressing shape". Conversely, the `lostOne` operation enables the applications to detect movement patterns such as: "an entity has left the addressing shape".

To illustrate these new operations, we program an application wherein a shopping trolley calculates its price with respect to product insertions and withdrawals. To detect product insertions and withdrawals, the shopping trolley defines a box that encompasses it. Each product publishes its price and the shape of the price is a point, which corresponds to the product location. Finally, we have two processes which are executed in parallel on the shopping trolley. The first process monitors the insertions of products and the second one monitors the withdrawals. In this way, price updates are synchronized on product insertions and withdrawals.

```
//First process
while(1) {
  readOnce(box_trolley, <"price", float p>);
  total += p;
}

//Second process
while(1) {
  lostOne(box_trolley, <"price", float p>);
  total -= p;
}
```

In these programs `box_shopping_trolley` represents the addressing shape of the shopping trolley. Each product publishes a tuple `<"price", p>`, with `p` corresponding to the price of the product.

4. Application

In this section, in order to illustrate how to program applications with Geo-Linda, we describe the Ubibus application [1]. Ubibus helps a visually impaired person to take the bus. A visually impaired person faces several difficulties when she wants to take the bus: she cannot see her bus arriving and cannot signal the driver to stop; when several bus lines stop at her bus stop, she cannot know if the bus that has just stopped is the one she wants to take. Ubibus tackles these problems by notifying the visually impaired person that her bus has just arrived, and by notifying the bus driver that a person wants to take his bus.

Ubibus is composed of three kinds of movement patterns: the arrivals and departures of pedestrians from bus stops, and the arrival of the bus at the bus stops. Each pedestrian, bus stop and bus executes a program, in order to detect these movement patterns.

The pedestrians' program starts by publishing a bus request. This request is a tuple which is composed by a request identifier, the line number and the direction. The shape of the tuple is a cylinder encompassing the pedestrian. The program then starts a `read` operation in order to detect the bus arrival. The addressing shape of this operation is a sphere centered on the pedestrian whose radius is 10 meters.

Pedestrians' program is the following:

```
out(cylinder, <"req_bus", 16, "beaulieu">);
read(sphere, <"bus_arrived", 16, "beaulieu">);
notify_pedestrian();
```

A bus stop monitors the number of pedestrians at the bus stop and publishes the stop request. The bus stop monitors the number of pedestrians with a program similar to the program of the shopping trolley. If the number of pedestrians is greater than zero, then the bus stop publishes a stop request for the bus. If all the pedestrians leave the bus stop, then the bus stop removes the stop request.

```
//First process
while(1) {
  readOnce(box_bus_stop,
           <"req_bus", 16, "beaulieu">) ;
  num_pedestrian++;
  if(num_pedestrian == 1)
    out(box_req_stop,
        <"req_stop", 16, "beaulieu">)
}

//Second process
while(1) {
  lostOne(box_bus_stop,
          <"req_bus", 16, "beaulieu">);
  num_pedestrian--;
  if(num_pedestrian == 0)
    drop(<"req_stop", 16,"beaulieu">)
}
```

A bus stop is a fixed synchronization point between the pedestrians and the bus. It detects the bus requests of the pedestrians and publishes a stop request for the corresponding bus. This is the bus stop which actually asks the bus to stop. If the pedestrian could publish stop requests, he could stop buses anywhere, like in the previous taxi application. This program illustrates an important programming construct: by using a third static entity, we can chose the synchronization place between two mobile entities.

The bus program is composed of two processes. The first process initially publishes a tuple that indicates to the pedestrian that his bus is arrived. The shape of this tuple to a sphere encompassing the bus. Then, the process starts a loop. Each iteration of this loop detects a stop request and notifies the driver that someone is waiting for his bus. The bus detects stop requests with a `take` in order to detect the waiting people and to reflect the state change of the bus stop: once the bus has arrived, the bus stop is not requesting a bus anymore.

The second process detects pedestrians that enter into the bus with `take` operations. Detecting pedestrians with `take` operations enables the bus to keep the physical memory consistent with the physical space, by deleting the bus requests. Indeed, a pedestrian that is inside a bus is not requesting a bus anymore. The addressing shapes of these `take` operations is a box included in the bus. In this way, the bus erases the bus requests only when the pedestrians are actually inside the bus.

```
//First process
out (sphere,
     <"bus_arrived", 16,"beaulieu"> );
while(1){
  take(cone, <"req_stop", 16,"beaulieu">);
  notify_driver();
}

//Second process
while(1) {
  take(box_bus,
       <"req_bus", 16, "beaulieu"> );
}
```

5. Evaluation

We implemented Geo-Linda in Java by extending the SPREAD framework [3]. We added the `take` , `readOnce` and `lostOne` operations, and the geometric addressing mode. Evaluations were done on Compaq HX4700 running the J9 JVM.

Implementation and detailed evaluation of the `take` operation are presented in [13]. The main results are a proof that atomic tuple transfers are not possible in the presence of message losses and a protocol which minimizes the number of non atomic take operations.

T	readOnce	lostOne
150 ms	60	50
300 ms	100	90
600 ms	180	160

Table 1. Number of tuples each operation can handle with respect to the poll period T

	Box	Cone	Cyl.	Point	Sector	Sph.	Tetra.
Box	23.0t	55.7t	50.8t	3.9t	19.3t	4.8t	18.7t
Cone		67.4t	65.1t	5.3t	24.0t	5.6t	24.8t
Cyl.			49.6t	5.6t	24.8t	5.2t	40.0t
Point				0.7t	4.0t	1.1t	3.4t
Sector					21.4t	24.2t	25.6t
Sph.						1.0t	11.1t
Tetra.							8.8t

Table 2. Complexity of the intersection tests with respect to a sphere / sphere test

The `readOnce` and `lostOne` operations poll the tuple space of the connected devices with a period of T, in order to detect a new or a lost tuple. A tuple is declared as lost if it is not present in two consecutive polls. This implementation raises a scalability issue with respect to the number of matching tuples these two operations can handle. Indeed, if the handling time of the received tuples is longer than T, the reaction time of the operations quickly increases, making them unusable. Therefore, we have evaluated the maximum number of tuples m that each operation can handle with respect to T (see table 1). With the current implementation, the programmer must evaluate the number of tuples involved in the application in order to determine the best T to be used. Future implementation will adapt T at runtime.

In order to help the application developer, we evaluated the relative complexity of the intersection tests between two volumes. Indeed, when an entity receives a reading request it must test whether the addressing shape of the operation intersects with the shape of the local matching tuples. Results are presented in table 2. To get results independent of the computing device, the durations presented in the table 2 are calculated with respect to the duration of an intersection test between two spheres. On the test platform, we have t=0.09ms. Using this table the application programmer can make compromises between the accuracy of used volumes and the duration of the intersection tests.

6. Related works

We presented in section 2 several systems to build physical shared data spaces. Geo-Linda adds a geometric addressing mode to the principle of physical data space. Traditional tuple spaces such as TSpaces [10] or JavaSpaces [11] do not support the mechanism of physical synchronization.

We already introduced LIME [15, 12] in section 2. LIME does not propose the `readOnce` operation, but a reaction mechanism which enables to obtain a similar semantic. The first major difference between LIME and Geo-Linda is the geometric addressing mode and the `lostOne` operation, which LIME does not support. Moreover, according to LIME's authors and this paper [2], LIME suffers from a scalability problem with respect to the number of cooperating devices. LIME locally replicates on each device the tuple space of the connected devices. This approach imposes a distributed transaction when an entity wants to withdraw a tuple from the tuple space. Geo-Linda does not replicate the local tuple space of connected devices and thus does not suffer from this scalability problem.

Römer and Schoch [16] present a programming model, enabling the creation of applications based on RFID detection. This programming model is event based and considers two kinds of events: arrival of new RFID in the reading area of the RFID reader; departure of an RFID. The `readOnce` and `lostOne` operations enable to detect similar events.

In [18], the authors present a programming model dedicated to cooperative artifacts. The programming model detects movement patterns with logic programming constructs. The state of a device, such as his location, and its direct environment are represented by a set of facts stored in a local fact base. The programmer defines rules that determine higher level facts, such as the proximity of two entities. These rules are evaluated when events occurs in the vicinity of the devices.

From the geometric point of view, our work has similarities with research works based on the virtual approach, which rely on a Geographical Information System (GIS) [14, 8]. A GIS stores its data in a spatial database [7]. A spatial database enables the programmer to issue queries including geometric conditions, such as "retrieve all the people which live in this area and who are older that 20 years". As we explained in the introduction, this approach raises a scalability problem, because it must keep up the spatial database consistent with the physical space. Geo-Linda does not rely on a central virtual model of the physical space. However, we can consider that each device contains a local model of the physical space, consisting of the shapes of the local tuples. This local model is defined relatively to the location of the device and moves with it, so it does not need any update to be kept consistent with the physical space.

Finally, in [9] the authors present a framework to develop media spaces. A media space is a set of volumes defined in the physical space. When the user enters a volume, an action, such as playing a sound, is triggered. This framework shares with Geo-Linda the triggering of actions based on the entrance of the user in a defined volume. However, this framework is not dedicated to the coordination of multiple devices, and cannot be used to detect movement patterns such as a meeting of devices, or the arrival of a new device in a volume.

7. Conclusion

In this paper we presented Geo-Linda, a physically distributed tuple space. Geo-Linda is dedicated to the programing of ubiquitous computing applications, involving movement patterns of objects and people, such as the arrival of a bus at a bus stop. Geo-Linda extends the notion of distributed tuple space with new reading operations, which enable us to detect new movement patterns, and a geometric addressing mode. The geometric addressing mode enables Geo-Linda to detect movement patterns precisely: tuples and reading operations are associated to a volume. An operation is released if its addressing volume intersects the volume of a matching tuple.

We illustrated Geo-Linda by programming Ubibus, an application to help visually impaired people to take the bus. Geo-Linda enables us to program applications by synchronizing program code on the occurrence of movement patterns. We first identify the movement patterns which compose this application and define the reading operations that will detect them. Then, we insert instructions after these reading operations. The execution of these instructions is triggered by the detection of the corresponding movement patterns.

References

[1] M. Banâtre, P. Couderc, J. Pauty, and M. Becus. Ubibus: Ubiquitous Computing to Help Blind People in Public Transport. In *Mobile HCI 2004*, pages 310–314, 2004.

[2] B. Carbunar, M. T. Valente, and J. Vitek. Coordination and mobility in CoreLime. *Mathematical Structures in Computer Science*, 14, 2004.

[3] P. Couderc and M. Banâtre. Ambient Computing Applications: An Experience with the SPREAD Approach. In *Annual Hawaii International Conference on System Sciences (HICSS'03)*, 2003.

[4] G. Cugola and G. Picco. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems, 2001.

[5] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[6] K. Goto and Y. Kambayashi. A New Passenger Support System for Public Transport using Mobile Database Access. In *VLDB*, 2002.

[7] R. H. Güting. An Introduction to Spatial Database Systems. *The VLDB Journal - The International Journal on Very Large Data Bases*, 3(4):357–399, 1994.

[8] M. Hope, T. Chrisp, and N. Linge. Improving Co-operative Working in the Utility Industry through Mobile Context Aware Geographic Information Systems. In *Proceedings of the eighth ACM International Symposium on Advances in Geographic Information Systems*, pages 135–140. ACM Press, 2000.

[9] R. Hul, B. Clayton, and T. Melamed. Rapid Authoring of Mediascapes. In *International Conference on Ubiquitous Computing (Ubicomp'04)*, 2004.

[10] T. J. Lehman, A. Cozzi, Y. Xiong, J. Gottschalk, V. Vasudevan, S. Landis, P. Davis, B. Khavar, and P. Bowman. Hitting the distributed computing sweet spot with tspaces. *Comput. Networks*, 35(4):457–472, 2001.

[11] S. Microsystems. Javaspaces. `http://www.sun.com/software/jini/`.

[12] A. Murphy, G. Picco, and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *International Conference on Distributed Computing System (ICDCS'01)*, pages 524–536, 2001.

[13] J. Pauty, P. Couderc, and M. Banâtre. Atomic Token Passing in the Context of Spontaneous Communications. In *Workshop on Applications and Services in Wireless Networks (ASWN'05)*, 2005.

[14] E. Peytchev and C. Claramunt. Experiences in Building Decision Support Systems for Traffic and Transportation GIS. In *Proceedings of the ninth ACM International Symposium on Advances in Geographic Information Systems*, pages 154–159. ACM Press, 2001.

[15] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME : Linda Meets Mobility. In *International Conference on Software Engineering (ICSE'99)*, pages 368–377, 1999.

[16] K. Römer and T. Schoch. Infrastructure concepts for tag-based ubiquitous computing applications. In *Workshop on Concepts and Models for Ubiquitous Computing (UbiComp'02)*, September 2002.

[17] G. Roussos, J. Tuominen, L. Koukara, O. Seppala, P. Kourouthanasis, G. Giaglis, and J. Frissaer. A Case Study in Pervasive Retail. In *Proceedings of the second international workshop on Mobile commerce*, pages 90–94. ACM Press, 2002.

[18] M. Strohbach, H.-W. Gellersen, G. Kortuem, and C. Kray. Cooperative Artefacts: Assessing Real World Situations with Embedded Technology. In *UBICOMP'04*, 2004.

[19] D. Touzet, F. Weis, and M. Banâtre. Sensing and filtering surrounding data: The persend approach. In *Mobile HCI Workshop on Mobile and Ubiquitous Information Access*, pages 283–297, 2003.

[20] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM, Back to the Real World, Special issue on Computer Augmented Environments*, 36(7):75–84, 1993.