Ambient computing applications: an experience with the SPREAD approach.

Paul Couderc and Michel Banâtre
INRIA Rennes / IRISA,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex - FRANCE
E-mail: {pcouderc, banatre}@irisa.fr

Abstract

Today’s we assist to the explosive development of mobile computing devices like PDAs and cell-phones, the integration of embedded intelligence (like Web server) in more and more common devices, and the proliferation of wireless communication technologies (IRdA, Bluetooth, IEEE 802.11, GPRS). All these trends contribute to move us closer to the ubiquitous computing world described by Mark Weiser.

But while the technology is here, applications, and more important, models and tools for designing future ambient computing systems are still rare. One of the first innovative concept of ubiquitous computing, context-awareness, is still hard to use and understand from a programming perspective. We think that the problem resides in the lack of system support: in traditional computing, operating system offers simple to use and easy to understand abstractions of computational resources. Ubiquitous computing involves an integration of “computing” into the real-world, which is a radically different environment for applications. We think that this environment requires new operating system services and abstractions. Because the real world is made of physical entities, “living” in the physical space, ambient computing software should be able to use abstraction representing such objects, in a simple way.

In this paper, we present a light framework to design ubiquitous computing software, called SPREAD. Unlike many approaches which hides too much of the real-world behind traditional computing abstraction, SPREAD defines programming abstraction based on the properties of the physical space. Hence, physical properties, like relative proximity, are used as implicitly in SPREAD as variable addressing in a computer memory. In SPREAD, application (or process) behavior can be “mechanically” driven, in the sense that actions flow can be directly dependent of physical mobility.

To support this concept, we introduce a programming and execution model allowing to design computing and information systems driven directly by arranging and moving physical objects in the space. We demonstrate the use of the model to implement a few practical applications, highlighting its simplicity and expression power.

1 Introduction

From a technological point of view, ubiquitous computing is already existing: PDAs, cellular phones, embedded Web servers, global and local wireless communications (GPRS, IEEE 802.11, Bluetooth, IRdA), positioning (Cell-ID, GPS, E-OTD, etc.) are all (relatively) common technologies. However the futuristic scenarios (ie, the applications) described by Weiser[22], like context-aware services, are still not common, except in laboratories.

We think that there are at least two important problems which explain this situation. First, ubiquitous computing is mainly about merging information system and the real world. This leads us to think that the success of ubiquitous computing depends more on the integration of concepts like context-awareness, in many applications (in new applications, as well as in existing popular applications like personal information managers, web browsers, search engines, . . .), than in the emergence of one killer application. There are different approaches to bridge real and virtual worlds and different understandings of the notions of context and context-awareness. Experimenting with many applications seems to us a pragmatic and very useful approach to help synthesizing the core structures, requirements and difficulties of ubiquitous computing systems. But this raises the second problem: easy prototyping.

To prototype small applications or extending existing ones, we adopted a minimalist (and low level) approach which implements one particular concept in flexible manner. This concept is called spatial programming: its principle is to abstract the physical space as if it were a memory, or a database, and considering mobility as a form of information addressing. Not all ubiquitous computing applica-
This paper describes some of our experiences with this system, called SPREAD for Spatial PRogramming Environment for Ambient computing Design. The first section discusses the motivations of our approach. The second section presents the principles of our approach. Then we introduce in the third section the spatial programming model, and we demonstrate its use on two applications. The fourth section details some of the implementations issues. The fifth section relates our approach to other works. Finally, we conclude the paper with some research perspectives.

2 SPREAD: a physical approach

One of the fundamental questions we try to answer about ubiquitous computing is “why is it so difficult to design and implement very simple application?” like the famous “hello, world!” An equivalent in ubiquitous computing could be for any physical object to “link” himself in the virtual world by creating an information item like “hello, I’m a the printer”. It is interesting to notice that this trivial application already raises some of the conceptual difficulties of ubiquitous computing: it is not clear to which entities the information item “hello, …” should be visible. To surrounding physical entities, or to any entities? Is the physical object “living”, by physically hosting the process that creates the information item, or is the object simply associated (for example by a localization system) with a software agent running elsewhere? Basically, we think that there is two main approaches: the first one is to represent the relevant information of the physical world in the information system. For example, by tracking the location of physical objects and building a geographic information system from the acquired data. We call this approach the “logical” one. And the other one is at the opposite: it considers the relevant information to be represented by the physical objects in the real world. We call the approach “physical”, and it is this one which has been adopted.

2.1 Physical nature of ubiquitous computing

Ubiquitous computing proposes to integrate computing system in the real world, in order to make them disappear from the user perception: by being aware of the state of the environment, ubiquitous (or invisible) computing systems allows seamless interactions with an enhanced (or augmented) environment. From a system perspective, we see a fundamental problem for these ubiquitous computing environments: a process no longer evolves in a virtual address space like in a traditional computer, but in the physical space, populated with physical objects.

We think that this physical nature should be reflected at the programming level, in order to implicitly take advantage of physical space properties, to structure the information system, and to drive the instruction flow.

2.2 Spatially structured information system

In a conventional computer memory, an information system is structured as pieces of information organized in a logical space. The following declaration defines a set of 100 books, organized sequentially in memory.

```c
struct book {
    char title[100];
    char author[20];
    int year;
} book_table[100];
```

A real library is organized very similarly, and we think that in ubiquitous computing environments, this physical organization should implicitly define the structure of the information system. More precisely, this means that in ubiquitous computing, the way of “declaring” the previous `book_table` should simply consist in arranging the physical books as shown in figure 1.

![Figure 1. A physical library](image)

2.3 Physical mobility as information addressing

Once the data structures are set up, we need to be able to use them. Scanning the previous book table would be done easily in a conventional program:

```c
for (i = 0; i < 100; i++) {
    use(book_table[i])
}
```

The equivalent in ubiquitous computing is not so obvious. As seen in the previous paragraph, the information is arranged in the physical space. Thus, it appears natural to interpret physical mobility as the way to address information. The process should then live in a mobile entity. The ubiquitous computing version of the previous table scanning process would consist in traversing the library linearly, like shown on figure 2.
Depending on the application, the process may either integrate human user decisions, or be completely automatic: in the library example, moving physically may be controlled by a human user (the process lives in a device embedded on the user), or by a mobile robot (the process lives in the robot, and controls its movements).

2.4 Action synchronization and environment perception

Ubiquitous computing involves multiple processes, because many entities “live” in the real-world, with potential processes associated with them. Hence, the problem of action synchronization and state perception must be addressed.

Some context-aware approaches are inspired from graphical interfaces programming, proposing event-driven programming: changes in environment state are perceived through sensors, which are synthesized as events, to which the program reacts. The problem is the same as for GUI programming: this imposes a flat structure to the program, uncoupled from its data processing goals, which basically loops for events and reacts to them through pre-defined handlers. The flow of actions is difficult to understand because, from the programmer perspective, actions are organized around the processing of the data structures (like computing the biggest element of a collection), while the event-driven structure makes the program structured around events processing.

This is similar to the complexity of distributed computing through message passing, which results in structuring the program around a communication protocol instead of the data structures. One of the most popular abstraction which eases parallel or distributed computing is the tuple-space, introduced in LINDA [11]. With the tuple-space abstraction, process actions are implicitly synchronized on the state of the data. We think that this notion of tuple-space is especially interesting to represent the state of the physical world to software, enabling implicit, or physical programming. The next section presents our model of tuple-space for ubiquitous computing.

3 Spatial information systems

In order to effectively support the physical nature of ubiquitous computing, we propose a tuple-space abstraction reflecting the logical state of the physical environment. This abstraction supports a spatial programming model, in which the structure and the control flow of the information system are directly dependent of physical entities and spatial relations between these entities.

3.1 Structure of spatial information systems

A spatial information system is made of a set of information items, each item filling a subset of the physical space. Thus, each point of the space is included in some (possibly zero) subsets of the space associated with an information item. The shape and size of this portion of space is arbitrary (but may be constrained to some shapes in a particular implementation, due to technological limitations).

In a similar way, any particular address of a computer memory corresponds to some information item. A record or an array may fills a particular address range in the memory. And more complex data structures like tree have more complex “shapes” in memory, filling for example non-contiguous address ranges.

3.2 Information addressing

From the previous description, we can store a set of information items in the physical space. We now explain how to solve the problem of addressing them in a computing process. The computing process is located at a given point in the physical space. This point determines the set of items usable by the process for an action (ex: an arithmetic operation), according to the portion of the space filled by each item. To read a different set of items, either the process has to move to a different point, or some items have to move in order to fill the subset of process addressable space.

One problem is that, at a given point, many items may be seen. Thus, spatial addressing is not sufficient to control what we want to address. Addressing information items according to their nature solves the issue. At present, we adopted the flexible approach of LINDA: each information item is structured as a tuple, which is a sequence of typed information items. For example, <10, ‘Peter’, -3.14> is a tuple where elements respectively are the integer 10, the string “Peter” and the real value -3.14.

Addressing information anonymously (without names) is done through “generative communication” [11], which
consists in specifying a pattern to match one or a set of tuples present in the tuple-space. A pattern is constituted of templates, where each element is either a type (meaning, match all values of this type), or a value (meaning, match this specific value). A pattern matching the previous tuple would be, for example, `<int, ‘Peter’, float>`.

The figure 3 shows an example of our “physical” tuple-space, made of tuples arranged in the space and occupying various sized spheres.

![Figure 3. Physical Tuple-Space](image)

### 3.3 Programming spatial information system

In a spatial information system, we program the applications with a minimalist API, yet very powerful, similar to LINDA. Four primitives are supported:

1. `out(tuple, area)`, publishes (inserts) a tuple in the tuple space. The tuple fills the physical space around the entity issuing the `out` operation. The shape and size of the portion space filled by the tuple is controlled by the `area` parameter. At the model level, any shape can be specified, but implementations have limitations, which depend on wireless communications and sensing technologies used.

2. `rd(pattern)`, returns a tuple matching the pattern, and covering a physical space containing the entity issuing the `rd`. The tuple is not removed from the tuple-space. Note that when several tuples match the pattern, any tuple from this set may be returned.

3. `capture(pattern)`, returns a set of tuples matching the pattern, and covering a physical space containing the entity issuing the `capture`. This primitive is a multiple tuples version of read (it returns all matching tuples). Like `rd`, `capture` blocks while no matching tuples are available.

4. `drop(tuple)`, allows the publisher of a tuple (the entity which issued the corresponding `out` operation) to remove the tuple from the tuple-space.

Unlike LINDA, we initially do not provide a destructive version of `rd` (called `in` in LINDA), which removes the matching tuple from the tuple-space. The reason is that in the semantic of our tuple-space, a tuple corresponds to a physical object; thus, removing the tuple means removing the physical object. However, to provide a way for a process to update a previously published tuple, the `drop` primitive can be used (but its use is restricted to the publisher of the tuple).

### 3.4 Applications

We now illustrate the use of the model to build two examples applications.

#### 3.4.1 Hitch-hiker application

The first application proposes to help hitch-hiking. A hitch-hiker publishes one or more destinations he is interested in (which is simply implemented by an `out` operation for each destination). A car accepting hitch-hikers just has to read pattern corresponding to hitch-hikers, which are tuples constituted of two strings: a prefix identifying a hitch-hiker ("go to"), followed by the destination. When the car approaches a hitch-hiker, the `rd` operation will match, un-blocking the next instruction which compares the requested destination to the car destination.

This application demonstrates the simplicity of implementing environment perception and contextual sensitivity using this model. It is summarized on figure 4.

![Figure 4. Hitch-hiker application](image)

More sophisticated applications based on the same pattern can be designed. Consider, for example, the negotiation of the price for a journey that you have to do with taxis in...
manycountries. By adding a price field to the tuples of the previous application, a client could specify the maximum price he agrees to pay for a given journey. The taxis would match on the journeys and prices they are interested in.

3.4.2 Collaborative rating

The second application demonstrates the spontaneous constitution of an information system through the physical space, where co-located physical entities contribute to a part of a computation. Consider the case of many small sport contests, where there is no computer assisted infrastructure to rank the competitors. The physical approach only requires the judges to be co-located around a chief-judge, and contribute their rating with a PDA as shown on figure 5. The chief-judge runs the “integration” part of the process, by computing the mean rating of all the visible ratings.

Some security issues are not address on the trivial code shown on figure 5. A simple way to ensure that only judges ratings are integrated in the final rating is to include an authentication key to the tuples.

4 Implementation issues

The model presented previously is implemented as a prototype to experiment ambient ubiquitous computing systems, called SPREAD. This section details some implementation issues of the model.

4.1 Architecture

The SPREAD engine is implemented around three main modules:

1. A tuple-space manager, which manages tuple storage and application requests through the primitives presented in section 3.3.
2. A tuple server, which offers remote access to the tuple-space for others tuple-space managers through simple get/put operations.
3. The discovery module, which manages the spontaneous aspect of the SPREAD tuple-space.

The prototype uses iPAQ 3600 serie PocketPC PDAs running WinCE 3.0 and IEEE 802.11b wireless LAN PCMCIA cards. The system is written in C, and the discovery module and remote tuple-space operations are based on the UDP protocol.

4.2 Managing tuple-space state

The tuple-space manager serves local application requests through the API previously defined. From the tuple-space manager point of view, tuples are separated into two classes: local tuples, created by local application with an out operation, and neighbor tuples, discovered from surrounding devices. This separation is transparent to applications.

The problem when implementing a tuple-space system is to design a strategy to propagate all relevant tuples to processes which are blocked on a read request. A simple strategy would be for each device to always broadcast its whole set of tuples. This is a write-driven strategy, because it is tuple writers which are responsible for propagating the tuples. Read-driven strategies work in the opposite way: tuple creation (calls to the out primitive) only results in local operations, while read operations (calls to rd or capture) may triggers remote operations.
We chose a read-driven strategy: this favors low bandwidth use when each process is interested in a few tuple patterns, which was the case with the applications we targeted for SPREAD. For example, the two applications described previously are only interested in reading one pattern: “go to” tuples for the hitch-hiker, and “rating” tuples for the collaborative rating application. Hence, when no process is trying to match tuples, bandwidth consumption is minimal.

The tuple-space manager maintains a list of read requests which are blocked. These entries are used by the discovery module to notify surrounding devices of pending read requests.

4.3 Handling read requests

When a process needs to read tuples, it issues either a \texttt{rd(pattern)} or \texttt{capture(pattern)}.

4.3.1 Rd

A call to \texttt{rd} creates an entry in the list of pending read, and blocks the caller, until the \texttt{match} procedure, which we explain later, decides to unblock it.

4.3.2 Match

\texttt{match} is an internal procedure which is used to determine whether a tuple matches some of the pending read requests (both local and remote). This procedure is executed either when a new tuple is created with a local \texttt{out}, or when the tuple server receives a remote tuple which it inserts into the neighbor tuples list. When a matching tuple is found, two cases must be considered: if the pending read request is local, it is released. If the pending request is remote, the tuple is sent to the remote tuple server.

4.3.3 Capture

\texttt{capture} operates in the same way as \texttt{rd}, but the decision to unblock the caller is more complex as we try to match all the tuples: when the \texttt{match} procedure releases the request, we do not unblock the calling thread until a certain delay (\texttt{capture delay}). This let enough time for surrounding devices to send their tuple matching pending read requests, and for \texttt{capture} to accumulate matching tuples.

4.4 Handling write requests

Tuples are initially created by an \texttt{out} operation issued by a process, which inserts an entry in the local tuple-space. The \texttt{match} procedure is executed after each \texttt{out} to release potentially blocked read requests, either local or remote.

4.5 The discovery procedure

An important feature of SPREAD is that it reflects implicitly the physical mobility at the programming level, through the state of tuple space. Hence, device discovery is a crucial aspect.

The discovery procedure consists in periodically broadcasting a presence message. This message includes a descriptor of the pending read requests of the device. Hence, each device periodically updates its list of pending read requests. Remote read requests are time-stamped, and cleaned periodically.

It is important to understand that, unlike other tuple-space systems, in SPREAD a read request may be blocked until a physical device approaches it (figure 7). It is this property which enables \textit{spatial programming}.

![Figure 7. spatial program](image)

4.6 Spatial addressing

In the general model, a tuple may cover an arbitrary shaped area in the space. However, implementing this property is difficult, to say the least. The only available technologies to implement tuple physical covers are radio wave or light (infra-red).

Infrared is interesting because it allows information (tuple) to be easily “aligned” on physical boundaries like rooms, corridors. However, infra red interfaces are sensitive to orientation and need sensors to be always exposed.

RF interfaces, like Bluetooth or IEEE 802.11, solve the limitations of infrared, but range, and shape definition of tuple cover are difficult to control. Using special antenna, reducing or increasing transmitter power or receiver sensitivity may help, but implementing these solutions is not always
possible. We think that the dynamic control of these parameters (through software) is an interesting open research issue.

Another interesting option that we are investigating is the use of relative local positioning: by using information from the physical layer like the strength of the received signal, it is possible to estimate the distance of the emitting object. Thus, by sending the logical range of a tuple specified in the `out` operation, it is possible for the receiver to filter out the tuples that it is not supposed to see.

5 Innovation and relation to other works

To understand an important difference between SPREAD and other systems, two opposite approaches of ubiquitous computing must be distinguished: the logical approach and physical approach.

5.1 Logical vs. Physical

The logical approach consists in building a logical representation of the physical space and physical entities. This approach is used, for example, in geographic information systems. The fundamental problem with this approach is that it imposes to maintain the representation of the physical space coherent with the reality, through tracking: for each movement of a physical entity, an update of the logical representation is needed.

The logical approach is like “emulating” the physical space in a computer system, and processing every context related operations at the level of the logical representation.

In the physical approach, the information system is implicitly constituted by physical entities. The structure and the content of the information system is derived from spatial properties of physical objects associated with information [5]. In such systems, control flow and information processing are directly done through physical objects placement and mobility like in a mechanical machine, except that, instead of processing physical forces, it processes information.

This approach has several advantages:

1. It is not needed to maintain the coherency between a logical representation of the physical world and the reality (which becomes a scalability problem when the number of physical objects associated with information system grows, or when the objects are highly mobile).

2. By structuring the information system on the physical world, the design of implicit computing applications is natural. We see this property as a strong argument in favor of this approach: it is far easier to design and program implicit computing applications with an underlying system supporting an implicit computing model, in the same way as programming a graphical user interface is easier with a graphical toolkit.

3. Because the information systems are physically spread over a great number of devices, the approach is potentially safer regarding privacy issues. There is no global access to information representing the spatial behavior of the user.

4. The systems developed with this approach are independent of any external network infrastructure, as they have their own communication capabilities. This is especially interesting for experimenting applications, as there is no impact on the networks. This is also a key feature for systems which have to operate without infrastructure support (hostile environment for example).

However, this approach requires to be able to integrate in each physical entity associated with information the necessary “intelligence” and communication capabilities, in order to autonomously participate in such a physical information system.

5.2 Related works

To our knowledge, no such physically-driven computing model has been previously proposed. The idea of exposing location in applications in order to structure the computing process around the notion of mobility and position in space is described in [19], but the perception and use of the location attribute is explicit: the need to request the position of a physical object qualifies this approach as a logical one.

The general idea of the computer systems fading into the environment is due to the pioneer Parctab project [20, 22], which introduced the concept of ubiquitous computing. Many works followed, exploring mainly two aspects of implicit computing: the first one is context-awareness, which consists in “guessing” the physical situation of a user to enable implicit access or annotation to information [1, 3, 4, 6, 16, 17]. The second aspect is made of multimodal user interfaces, which enables implicit interactions with computer systems through natural senses. Some examples includes [7, 8, 9, 10, 14].

The spontaneous aspect of ubiquitous computing has been studied in particular by the spontaneous information systems project [2, 21], and Proem [13]. These systems trigger implicit information exchanges when devices encounter each others. SPREAD provides a spatial computing model based on this concept, which makes it more general.

Spatial databases [12], which are database systems optimized to store and manage geographic or spatial information have similarities with our work: in both cases, the
information system depends on spatial or geometric properties. But spatial databases are logical approaches: they build a logical representation of the physical space, unlike our model of spatial programming which relies implicitly on the physical space.

Finally, our work shares similarities with the LINDA programming model [11] because we rely on the concept of tuple-space. The important difference between the model we consider and LINDA, or other tuple-space systems with similar semantics (Sun’s JavaSpace [15], IBM’s T-Space [23]), is the following: when an entity issues a matching request, only the tuples filling the physical space containing the entity will match. LINDA semantic assumes a global tuple-space shared by all the entities, where only patterns contribute to information addressing. There is no implicit coupling between the tuple-space state and the physical space.

One work is closed to our: LIME [18]. LIME transparently alters the tuple-space state according to processes location. A tuple created by one process is visible to other process if they are co-located, meaning that a communication link between them is available. If this communication link is only available when hosting devices are close one from each other, the semantic is very similar to SPREAD. There is however a conceptual difference: the SPREAD model assumes computing in the physical space. The physical space is considered as a kind of computer memory, with information covering a specified portion of the (physical) space. tuples and the LINDA model provides an easy way to implement this notion of physical computing, which is the core idea of SPREAD. To highlight this point, remember that SPREAD always associate tuples with physical objects: this explains why SPREAD tuple-space does not support the destructive in operation, which would mean that a physical object may physically destroys another object. SPREAD tuples, at least at the model level, cover a specified region of the physical space. This is different from LIME tuples, which are not associated with a region of the space. In LIME, a tuple can be addressed by a process when this latter is co-located with the component which created the tuple.

6 Conclusion

This paper presented some of our experiences implementing ubiquitous computing systems, using the concept of spatial programming. An interesting aspect of this approach is its focus on allowing implicit computing at the programming level, unlike many works which limit this aspect only to the user.

With a computing model based on the physical world, programming ubiquitous computing applications is more natural. Building information systems by simply arranging and moving objects in the physical space is clearly in the trend initiated by Weiser with the concept of ubiquitous computing. A slogan for spatial programming would be to say that the real world is the computer.

However, realistic use of this concept raised important issues which need to be addressed. Some of them are the following:

First, spatial addressing is still difficult to finely implement using the existing technologies.

Second, the discovery procedure is highly dependent on physical parameters. A careful analysis of the relation between parameters like device speed, communication interface range, and discovery protocol parameters is required. This particular issue is the object of current active work.

Finally, with so closed integration of information systems and physical world, privacy and anonymity become important problems, especially at the application level.

Besides system issues, the main research perspective is to explore new applications which could take advantage of the SPREAD approach, eventually leading to enhancements in the spatial programming model.

References


