Programming Context-Aware Pervasive Computing Applications with TOTA

Marco Mamei[1,2], Franco Zambonelli[2], Letizia Leonardi[1]

[1]*Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia*
Via Vignolese 905 – Modena – ITALY
[2]*Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia*
Via Allegri 13 – Reggio Emilia – ITALY

{ mamei.marco, franco.zambonelli, letizia.leonardi }@unimo.it

Abstract

Pervasive computing calls for suitable programming models and associated supporting infrastructures to deal with large software systems dived in complex and dynamic network environments. Here we present TOTA, a new approach for the development of pervasive computing applications. TOTA proposes relying on tuple-based information to be spatially diffused in the network on the basis of some application-specific propagation rule, to be exploited by application agents to achieve context-awareness and to effectively coordinate with each other. As shown with the help of a case study scenario, TOTA, while promoting a simple programming model, is effective to facilitate access to distributed information, navigate in complex networks, and enforce complex coordination activities in an adaptive way.

***Keywords**: Pervasive Computing, Distributed Programming Models, Tuples, Coordination, Mobility.*

1. Introduction

Computing is becoming intrinsically ubiquitous and mobile [EstC02]. Computer-based systems are going to be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of interacting with each other in the context of complex distributed applications, e.g., to support our cooperative activities [Mts2002], to monitor and control our environments [Bor02], and to improve our interactions with the physical world [MamLZ02]. Also, since most of the embeddings will be intrinsically mobile [PicMR00], as a car or a human, distributed software processes and components (from now on, we adopt the term "agents" to generically indicate the active components of a distributed application) will have to effectively interact with each other and effectively orchestrate their activities despite the network and environmental dynamics induced by mobility.

The above scenario introduces peculiar challenging requirements in the development of distributed software systems: *(i)* since new agents can leave and arrive at any time, and can roam across different environments, applications have to be *adaptive*, and capable of dealing with such changes in an adaptive and unsupervised way; *(ii)* the activities of the software systems are often contextual, i.e., strictly related to the environment in which the systems execute (e.g., a room or a street), whose characteristics are typically a priori unknown, thus requiring to dynamically enforce *context-awareness*; *(iii)* the adherence to the above requirements must not clashes with the need of promoting a *simple programming model* requiring light supporting infrastructure, possibly suited for resource-constrained and power-limited devices.

Unfortunately, current practice in distributed software development are unlikely to effectively address the above requirement: *(i)* application agents are typically strictly coupled in their interactions (e.g., as in message-passing models), thus making it difficult to promote and support spontaneous interoperations; *(ii)* agents are provided with either no contextual information at all or with only low-expressive information (e.g., raw local data or simple events), difficult to be exploited for complex coordination activities; *(iii)* due to the above, the results is usually in an increase of both application and supporting environment complexity. The approach we propose in this paper build on the lessons of uncoupled coordination models like event-based [CugFD01] and tuple space programming [GelC92, CabLZ02] and aims at providing agents with effective contextual information that – while preserving the lightness of the supporting environment and promoting simplicity of programming – can facilitate both the contextual activities of application

agents and the definition of complex distributed coordination patterns.

In TOTA ("Tuples On The Air"), all interactions between application agents take place in a fully uncoupled way via tuple exchanges. However, unlike in traditional tuple-based model, there is not any notion like a centralized shared tuple space. Rather, tuples are some how injected into the network and, in the network, can propagate and diffuse accordingly to a tuple-specific propagation pattern, automatically re-adapting such patterns accordingly to the dynamics changes that can occur in the network. All interactions are mediated by these distributed tuples, which can express both information explicitly provided by other agents or a local view of some global property of the network. As we will show later in this paper, this facilitate the ease definition of very complex coordination activities. To take a metaphor with the physical world, we can imagine the tuples as physical fields like the electromagnetic one. Physical particles do neither interact directly with each other nor they explicitly perceive the environment in which they are situated; rather, global motion patterns emerge as an expression of the local motion activities driven by the locally perceived field.

The contribution of this paper is to motivate and present the key concepts underlying TOTA, and to show how it can be effectively exploited to develop in a simple way adaptive and context-aware distributed applications, suitable for the dynamics of pervasive computing scenarios. To this end, the following of this paper is organized as follows. Section 2 introduces the case study scenario and motivates the need for a novel approach to application development. Section 3 overviews the TOTA approach. Section 4 details how to program distributed applications in TOTA. Section 5 exemplifies how the case study can be actually programmed in TOTA. Section 6 discusses related works. Section 7 concludes and outlines future works.

2. Motivations and Case Study

To sketch the main motivations behind TOTA, we introduce here a simple case study scenario and try to show the inadequacy of traditional approaches in this context.

2.1. Case Study Scenario

Let us consider a big museum, and the variety of people moving within it. These can include tourists visiting the museum as well as security guards in charge of monitoring it. We assume that each of these persons is provided with a wireless-enabled computer assistant (e.g., a PDA). Also, it is realistic to assume the presence, in the museum, of a densely distributed network of computer-based devices, associated art pieces, alarm systems, climate conditioning systems, etc. Such devices can be exploited for both the sake of monitoring and control, as well as for the sake of providing tourists and security guards with information helping them to achieve their goals. For tourists, such goals may include retrieving information about art pieces, effectively orientate themselves in the museum, and meeting with each other (in the case of organized groups); for security guards, these may include discovering anomalies in the museum and coordinate their movements and positions with each other to do their cooperative work in an efficient way. In the following, we will concentrate on two specific representative problems: *(i)* how users can gather and exploit information related to an art piece they want to see; *(ii)* how security guards can organize their respective position so as to properly monitor the museum.

In any case, whatever specific application problem has to be addressed in the above scenario, it should meet the requirements identified in the introduction. *(i) Adaptivity*: tourists and security guards move in the museum, new tourists are likely to come and go at any time, new security guards can arrive to support the existing team, art pieces can be moved around the museum during special exhibitions of during restructuring works. Thus, the topology of the overall network can change with different dynamics and for different reasons, all of which have to be preferably faced without human intervention. *(ii) Context-awareness*: as the environment (i.e., the museum map and the location of art pieces) may not be know a priori (tourists can be visiting the museum for the first time and security guards may have been temporarily hired), and it is also likely to change in time (due to restructuring and exhibitions), application agents should be dynamically provided with contextual information helping their users to move in the museum and to coordinate with each other without relying on any a priori information; *(iii) Simplicity:* PDAs may have limited battery life, as well as limited hardware and communication resources. This may require a light supportive environment and the need for applications to achieve their goal with limited computational and communication efforts.

We emphasize the above sketched scenario exhibits characteristics that are typical of a larger class of pervasive computing scenarios. Among the others, traffic management and manufacturing control systems [MamLZ02], mobile robots and sensor networks [EstC02].

2.2. Inadequacy of Traditional Approaches

Most coordination models and middleware used so far

in the development of distributed applications appears inadequate in supporting coordination activities in pervasive computing scenarios, as the one in the previous subsection.

In *direct communication models*, a distributed application is designed by means of a group of agents that are in charge of communicating with each other in a direct and explicit way. Systems like Jini [Jini], as well as FIPA systems [BelPR01], support such a direct communication model. One problem of this approach is that agents, by having to interact directly with other, can hardly support the openness and dynamics of pervasive computing scenarios: explicit and expensive discovery of communication partners typically supported by some sort of directory services, has to be enforced. Also, agents are typically placed in a “void” space: the model, *per se*, does not provide any contextual information, agents can only perceive and interact with (or request services to) other agents, without any higher level contextual abstraction. Thus, each agent has become context aware by explicitly requesting contextual information to local services. In the case study scenario, visitors and security guards have to explicitly discover location of art pieces and of other security guards. Also, to orchestrate their movements, security guards must explicitly keep in touch with each other and agree on their respective movements via direct negotiation. These activities require notable computational and communications efforts and typically ends up with ad-hoc solutions – brittle, inflexible, and non-adaptable – for a contingent coordination problem.

Shared data-space models exploit localized data structures in order to let agents gather information and interact and coordinate with each other. These data structures can be hosted in some centralized data-space (e.g., tuple space), as in JavaSpaces [FreHA99], or they can be fully distributed over the nodes of the network, as in MARS [CabLZ02]. In these cases, agents are no longer strictly coupled in their interactions, because tuple spaces space mediates interactions and promotes uncoupling. Also, tuple spaces can be effectively used as repositories of local, contextual information. Still, such contextual information can only represent a strictly local description of the context that can hardly support the achievement of global coordination tasks. In the case study, one can assume that the museum provides a set of data-spaces, storing information such as nearby art pieces as well as messages left by the other agents. Visitors can easily discover what art pieces are nearby them, but to locate a farthest art piece they should query either a centralized tuple space or a multiplicity of local tuple spaces, and still they would have to internally merge all the information to compute the best route the target. Security guards can build an internal representation of the actual formation by storing tuples about their presence and by accessing several distributed data-spaces. However, the availability of such information does not free them from the need of negotiating with each other to orchestrate movements. In other words, despite the availability of some local contextual information, a lot of explicit communication and computational work is still required to the application agents to effectively achieve their tasks.

In *event-based publish/subscribe models*, a distributed application is modeled by a set of agents interacting with each other by generating events and by reacting to events of interest. Typical infrastructures rooted on this model are: Jedi [CugFD01] and Jini Distributed Events [Jini]. Without doubt, an event-based model promotes both uncoupling (all interactions occurring via asynchronous and typically anonymous events) and a stronger context-awareness: agents can be considered as embedded in an active environment able of notifying them about what is happening which can be of interest to them (as determined by selective subscription to events). In the case study example, a possible use of this approach would be to have each security guard notify its movements across the building to the rest of the group. Notified agents can then easily obtain an updated picture of the current formation in a simpler and less expensive way than required by adopting shared data spaces. However, such information still relies on agents for the negotiating the coordinated movements and does not alleviate their computational tasks (i.e., in the case study, security guards still have to explicitly negotiate their movements).

3. The Tuples on the Air Approach

The definition of TOTA is mainly driven by the above considerations. It gathers concepts from both tuple space approaches [GelC92, CabLZ02] and event-based ones [CugFD01] – thus preserving uncoupling in interactions – and extends them to provide agents with abstract – simple yet effective – representations of the context. The goal is to enable specific coordination activities to be implicitly and with minimal effort realized by application agents, and to be automatically adapted to the dynamics of the execution scenarios.

3.1. Overview

TOTA proposes relying on distributed tuples for both representing contextual information and enabling uncoupled interaction among distributed application agents. Unlike traditional shared data space models, tuples are not associated to a specific node (or to a specific data space) of the network. Instead, tuples are injected in the network and can autonomously propagate and diffuse in the network accordingly to a specified

pattern (see Figure 1). Thus, TOTA tuples form a sort of spatially distributed data structure able to express not only data to be transmitted between application agents but, more generally, some property of the distributed environment.

To support this idea, TOTA is composed by a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. Each TOTA node holds references to a limited set of neighboring nodes. The structure of the network, as determined by the neighborhood relations, is automatically maintained and updated by the nodes to support dynamic changes, whether due to nodes' mobility or to nodes' failures. The specific nature of the network scenario determines how each node can found its neighbors: e.g., in a MANET scenario, TOTA peers are found within the range of their wireless connection; in the Internet they can be found via an expanding ring search. More details on the architecture and implementation of the TOTA middleware can be found in a companion paper [MamZL02].

Upon the distributed space identified by the dynamic network of TOTA peers, each agent is capable of locally storing tuples and letting them diffuse through the network (see Figure 2). Tuples are injected in the system from a particular node, then they spread hop-by-hop accordingly to their propagation rule. Accordingly, a tuple in TOTA is defined in terms of a "content", and a "propagation rule".

T=(C,P)

The content *C* of a tuple is basically an ordered set of typed fields representing the information carried on by the tuple. The propagation rule *P* determines how a tuple should be distributed and propagated in the TOTA network. This includes determining the "scope" of the tuple (i.e. the distance at which such tuple should be propagated and possibly the spatial direction of propagation) and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rules can determine how tuple's content should change while it is propagated. Tuples do not necessarily have to be distributed replicas: by assuming different values in different nodes, tuples can be effectively used to build a distributed data structure expressing some kind of spatial/contextual information. We emphasize that the TOTA middleware supports tuples propagation actively and adaptively: by constantly monitoring the network local topology and the income of new tuples, the middleware automatically re-propagates tuples as soon as appropriate conditions occur. For instance, when new peers get in touch with a network, TOTA automatically checks the propagation rules of the already stored tuples and eventually propagates the tuples to the new peers. Similarly, when the topology changes due to peers' movements, the distributed tuple structure automatically changes to reflect the new topology.

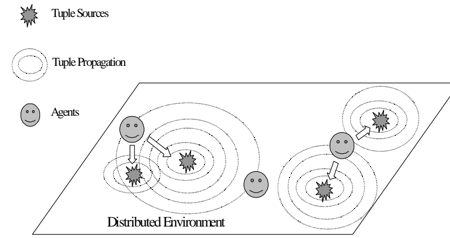

Figure 1: **The General Scenario of TOTA: active software agents, embedded in a distributed networked environment can inject tuples in the system that autonomously propagate or sense tuples present in their area.**

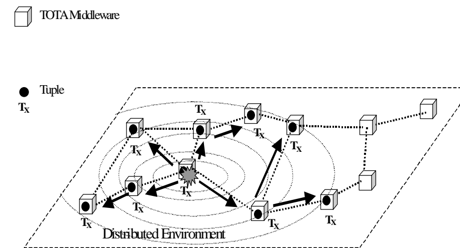

Figure 2: **The TOTA Network, in which tuples propagates by means of a multi-hop mechanism.**

Form the application agents' point of view, executing and interacting basically reduces to inject tuples, perceive local tuples, and act accordingly to some application-specific policy. Software agents on a TOTA node can inject new tuples in the network, defining their content and their propagation rule. They have full access to the local content of the middleware (i.e., of the local tuple space), and can query the local tuple space – via a pattern-matching mechanism – to check for the local presence of specific tuples. In addition, agents can be notified of locally occurring events (i.e., changes in tuple space content and in the structure of the network neighborhood). In TOTA there is not any primitive notion of distributed query. Still, it is possible for a node inject a tuple in the network and have such distributed tuple be interpreted as a query at the application-level, by having other agents in the network react to the income of such tuple, i.e., by injecting a reply tuple propagating towards the enquiring peer.

As an additional note, the possibility of propagating tuples without storing them in the propagation nodes enables TOTA to be used not only as a distributed

information repository, but also as a distributed event dispatcher [CugFD01].

3.2. The Case Study in TOTA

Let us consider the case study introduced in Section 2. We recall that we assume that the museum is properly instrumented with a reasonably dense number of wireless TOTA peers, e.g., associated with museum rooms and corridors as well as with art pieces, and that visitors and security guards are provided with wireless enabled PDAs running the TOTA middleware. All these devices, by connecting in ad-hoc network, define the structure of the TOTA space, which is likely to globally reflect the topology of the museum's plan.

The first problem we face is that of enabling a visitor to discover the presence and the location of a specific art piece. TOTA makes this very simple, and let us envision two possible solutions. *(i)* Each art piece in the museum can propagate a tuple having as a content *C* its description, its location, and a value specifying the distance of the tuple from its source (i.e., of the art piece itself). In other words:

C= (description, location, distance)

P=(propagate to all peers hop by hop, increasing the "distance" field by one at every hop)

Then, any visitor, by simply checking its local TOTA tuple space, can discover where the art piece is located and, by following backwards the tuple up to its source, can easily reach it without having to rely on any a priori global information about the museum plan. *(ii)* As an alternative, we could consider that art pieces do not propagate a priori any tuple, but they can sense the income of tuples propagated by visitors – and describing the art piece they are looking for – and are programmed to react to these events by propagating backward to the requesting visitors their own location information.

The second problem we consider involves the security guards to move and monitor the museum in a coordinated way, i.e., according to a specific formation in which they have to preserve a specified distance from each other. To this end, we can take inspiration from the work done in the swarm intelligence research [Bon99]: flocks of birds stay together, coordinate turns, and avoid each other, by following a very simple swarm algorithm. Their coordinated behavior can be explained by assuming that each bird tries to maintain a specified separation from the nearest birds and to match nearby birds' velocity. To implement such a coordinated behavior in TOTA and apply it in our case study, we can have that each security guard generates a tuple *T=(C,P)* with following characteristics:

C= (FLOCK, peerName,val)

P= ("val" is initialized at 2, propagate to all the peers decreasing by one in the first two hops, then increasing "val" by one for all the further hops)

Thus creating a distributed data structure in which the *val* field assumes the minimal value at specific distance from the source (e.g., 2 hops), distance expressing the intended spatial separation between security guards. For a tuple, the *val* field assumes a distribution approaching the one showed in Figure 3-left. The TOTA middleware ensures dynamic updating of this distribution to reflect peers' movements. To coordinate movements, peers have simply to locally perceive the generated tuples, and to follow downhill the gradient of the *val* fields. The result is a globally coordinated movement in which peers maintain an almost regular grid formation (see Figure 3-right).

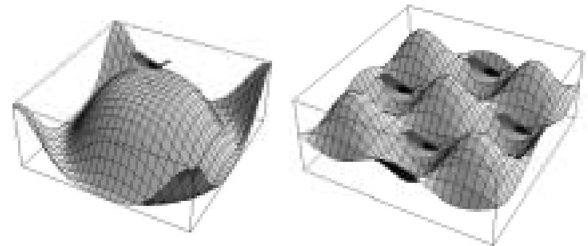

Figure 3. **Distribution of a single FLOCK tuple (left); Regular formation of flocking peers (right)**

3.3. The Concept of Space in TOTA

The type of context-awareness promoted by TOTA is strictly related to spatial-awareness. In fact, by creating an overlaid, distributed data structure, TOTA tuples intrinsically provides a notion of space in the network. For instance, a tuple incrementing one of its fields as it gets propagated identifies a sort of "structure of space" defining the network distances from the source.

However, TOTA allows dealing with spatial concepts in a much more flexible way. Although at the primitive level the space is the network space, and distances are measured in terms of hops from peer to peer, it is possible to exploit more physically-grounded concepts of space. These may be required by several pervasive computing scenarios in which applications agents need to interact with and acquire awareness of the physical space For instance, if some sort of localization mechanism, whether GSP or beacon-based triangulation [HigB01], is available to peers, then tuples propagation rules can also be expressed exploiting the available spatial information. Specifically, one can bound the propagation of a tuple to a portion of physical space by having the propagation procedure – as the tuple propagates from node to node – check in the local tuple space the local spatial coordinates, so as to deciding weather to further propagate the tuple or not.

In addition, one could think at mapping the peers of a TOTA network in any sort of virtual space. This space, that must be supported by an appropriate routing mechanism allowing distant peers to be neighbors in the virtual space (e.g., the normal IP protocol) can then be used to propagate tuple so as to realize content-based routing policies, as in CAN [Rat01] and Pastry [RowD01].

For the sake of simplicity, and although our case study would be likely to require physical-space awareness, in our examples we will continue to refer to the "network" space, and to distances as network-distances, i.e., number of hops.

4. TOTA Programming

When developing applications upon TOTA, one has basically to know:

- what are the primitive operations to interact with the middleware;
- how to specify a tuple and its propagation rule;
- how to exploit the above to code agent coordination.

4.1. TOTA Primitives

TOTA is provided with a simple set of primitive operations to interact with the middleware. The main operation:

```
public void inject (Tuple tuple)
```

is used to inject the tuple passed as an argument in the TOTA network. Once injected the tuple starts propagating accordingly to its propagation rule (embedded in the tuple definition).

The following two primitives give access to the local list of stored tuples:

```
public ArrayList read (Tuple template)
public ArrayList delete (Tuple template)
```

The read primitive accesses the local TOTA tuple space and returns a collection of the tuples locally present in the tuple space and matching the template tuple passed as parameter. The delete primitive extracts from the local middleware all the tuples matching the template and returns them to the invoking agent.

In addition, two primitives are defined to handle events:

```
public void subscribe (Tuple template,
                                   String reaction)
public void unsubscribe (Tuple template)
```

These primitives rely on the fact that any event occurring in TOTA (including: arrivals of new tuples, connections and disconnections of peers) can be represented as a tuple. Thus: the subscribe primitive associates the execution of a reaction method in the agent (whose name is specified as second parameter) in response to the occurrence of events matching the template tuple passed as first parameter. The unsubscribe primitives removes matching subscriptions.

To clarify the above concepts let us consider the following simple application agent:

```
public class QueryAgent {
 TotaMiddleware totam;
 // reference to middleware
 // initialized in construction
 public QueryAgent(TotaMiddleware totam) {
  this.totam = totam;   }

 // agent body
 public void start() {
  totam.inject(new QueryTuple("Monna Lisa"));
  totam.subscribe(new DescrTuple
                    ("Monna Lisa",null),
                      "displayReaction");  }

 // code of the reaction
 public void displayReaction(DescrTuple event) {
System.out.pritnln("Monna Lisa in" +
                              event.location); }
}
```

After construction, the agent performs just two simple operations: it injects in the network a tuple of class *QueryTuple* (this tuple and its propagation will be better described in 5.1, but basically it is used to propagate an interest for a particular art piece). Then it subscribes to the income of all the *DescrTuple* tuples (which are assumed to describe an art piece and its location) having as the first field "*Monna Lisa*", the agent in the Monna Lisa painting is expected to generate. The associated reaction diplayReaction is executed on receipt of such tuple to prints out the content of the received event tuple.

4.2. Specifying Tuples

TOTA, implemented in Java, relies on an object oriented tuple representation. The system is preset with an **abstract class Tuple** providing a general framework for tuples programming. Basically this class provides a general mechanism (propagate method) implementing – with the support of the middleware – a general purpose breadth first, expanding ring propagation. In this class four abstract methods, controlling the tuple propagation, its content update and its behavior have been defined:

```
public abstract boolean decidePropagation();
public abstract void doAction ();
public abstract Tuple changeTupleContent();
public abstract boolean decideStore ();
```

These methods must be implemented when subclassing from the abstract class Tuple to create and instantiate actual tuples. This approach is very handy when programming new tuples, because there is no need to re-implement every time the propagation mechanism from scratch, but it allows to customize the same breadth first, expanding ring propagation for different purposes.

In particular the overall structure of the propagation

process is the following:

```
if(decidePropagation()) {
 doAction();
 updatedTuple = changeTupleContent();
 if(decideStore()) {
     totam.store(updatedTuple);
    }
}
```

All these methods acts on the tuple itself (the "this" in OO terminology) can access the local TOTA middleware in which they execute (i.e., in which a propagating tuple has arrived).

The first method (*decidePropagation*) is executed upon arrival of a tuple on a new node, and returns true if the tuple has to be propagated in that middleware, false otherwise. So for example an implementation of this method that returns always true, realize a tuple that floods the network because it is propagated in a breadth first, expanding ring way to all the peers in the network. Vice-versa an always false implementation creates a tuple that does not propagate, because the expanding ring propagation is blocked in all directions. In general more sophisticate implementations of this method are used to narrow down tuples spreading to avoid network flooding.

The second method (*doAction*) allows the tuple to undertake some operations on the local middleware in which it is going to be propagated. Basically, through the use of this method a tuple can access the TOTA API provided by the middleware, and thus read, inject or delete tuples.

The third method (*changeTupleContent*) creates a new tuple that updates the previous one in the propagation process. This method is used because a tuple can change its content during the propagation process. Basically before propagating a tuple to a new node, the tuple is passed to this method that returns an eventually modified version of it, to be propagated. So for example, if the tuple content is an integer value and this method returns a tuple having that value increased by one, we are creating a tuple whose integer content increases by one at each propagation hop.

The fourth method (*decideStore*) returns true if the tuple has to be stored in the local middleware, false otherwise. This method can be used to create transient tuples, i.e., tuples that roam across the network like messages (or events) without being stored.

Eventually, once the above process determined that a tuple has to be locally stored, a method of the middleware with restricted access (the *store* method) is invoked to store a tuple in the local tuple space.

To clarify these points let's consider an example that will be used also when addressing the case study in 5.2. Let's create a tuple *FlockingTuple* that realize the flocking tuple introduced in Subsection 3.2.

```
public class FlockingTuple extends Tuple {
public int val = 2;
public int hop = 0;
// the attributes of the tuple represent,
// orderly, the fields of the tuple

public boolean decidePropagation()
{ return true;
}
```

The *decidePropagation* method returns always true, because the flocking tuple should be spread to the whole network. This is in order to create just one large flock, rather than some separate flocks.

```
public void doAction() {
 }
public boolean decideStore() {
 return true;
 }
}
```

The two methods above simply mean that the tuple has no specific actions to do while propagating, and that it must be stored in every node.

```
public Tuple changeTupleContent ()  {
 int newVal = this.val;
 int hop = this.hop;
 /* flocking shape */
 if(hop <= 2) newVal--;
 else if(hop >2) newVal++;
 FlockingTuple flock = new FlockingTuple();
 flock.hop=this.hop+1;
 flock.val=newVal;
 return flock;
}
```

This method changes the tuple being propagated by increasing its *hop* value by one at every hop. While *val* value decreases for the first two hops then increases.

5. Detailing the Case study

5.1. Gathering Contextual Information

In this section we are going to give more detail about the case study of gathering contextual information. Following the second approach sketched in 3.2, we consider a solution in which art pieces are programmed in order to sense the income of query tuples propagated by visitors and to react by propagating backward to the requesting visitors their location information.

Basically the software running on visitors' PDA is the *QueryAgent* described in 4.1, while software running on an art piece embedded computer is the following *ArtAgent*.

```
public class ArtAgent {
    TotaMiddleware totam;
    String description;
```

```
// agent construction
public ArtAgent(TotaMiddleware totam,
                              String descrption)
  { this.totam = totam;
    this.description=description; }

  // agent body
  public void start() {
  // get ready to answer to queries
  totam.subscribe(new
       QueryTuple(description,null),
                        "answerReaction");
   }

   public void answerReaction(Tuple event) {
    // upon receipt of a query
    // propagate backward a DescrTuple
     DescrTuple t = new DescrTuple(
         this.description, totam.Location());
     totam.inject(t);
     }
   }
  }
```

Basically this agent is identified by a description, representing the art piece and its behavior is to subscribe to the tuples querying for itself.
The reaction to such an event, is to inject an answering tuple that simply follows backward the query tuple to reach the visitor agent issuing the request.

We will not show the complete code of the two tuples involved, but just a simple description and a fragment of the code of answering tuple:

Query Tuple
C = (description,val)
P = (propagate flooding the network, until a matching peer has been found. Increase val by one at every hop)
AnswerTuple
C = (description,location)
P = (propagate following downhill the val of the associated QueryTuple, incrementing distance value by one at every hop)

Let's briefly see a fragment of code of the answering tuple: basically the DescrTuple follows downhill the path laid down by the QueryTuple to reach the destination.

```
  public boolean decidePropagation()
   {
    if(!isDownhill(this.val))
     return false;
    return true;
   }
  public boolean isDownhill(int refVal)
   {
    int vval = getVal();
    if(vval == -1)
     return false;
    return  vval < refVal;
   }
  public int getVal()
   {
    Tuple ttf = totaMw.getTuple(tupleToFollow);
    if(ttf != null && (ttf instanceof
                            QueryTuple))
     return ((QueryTuple)ttf).val;
    else
     return -1;
   }
```

In this fragment we see the *decidePropagation* method of the DescrTuple: it just checks if in the node the tuple is going to be propagated, the associated QueryTuple's val decreases.

5.2. Flocking

The algorithm followed by flocking agents is very simple: they simply have to determine the closest peer, and then move by following downhill that peer's flocking tuple. To achieve this goal each agent will evaluate the gradient of the flocking tuple it is going to follow, by comparing tuple's values in its neighborhood.

Other than the flocking tuple described in section 4.2, two other tuples have to be defined for this application. In order to keep the description simple, we will not show the code for these other tuples, but we just present a short description:

GradFlock is a tuple that propagates to all the peers within one hop distance from its source, having the following content:

$C=(X,Y,peerName_1,val_1,peerName_2,val_2,\ldots,peerName_n,val_n)$

X,Y are the coordinate where the source peer is located (we can imagine that these coordinates are obtained by some kind of localization mechanism like a GPS), The GPS system would be connected to the TOTA middleware to ensure that the coordinates in the tuple always provides the actual peer's location.

$peerName_i$ are the names of all the peers in the flock

val_i are the associated flocking tuples' values evaluated on the peer the emits the *GradFlock* tuple.

This tuple is needed because each flocking peer determines in which direction to move by evaluating the gradients of the flocking tuples, and such a gradient can only be determined by the knowledge of the tuple's values in the peer's neighborhood.

QueryTuple is a tuple that propagates to all the peers within one hop distance from its source and having the following content:

$C=(peerName_1,peerName_2,\ldots,peerName_n)$

Where $peerName_i$ are the names of all the peers in the flock.

Moreover, the *doAction* method of this tuple (discussed in 4.2) instructs the foreign middleware to inject a *GradFlock* tuple in the system considering the

peers in the content of the *QueryTuple*, thus realizing a query-response mechanism.

```
public class FlockingAgent {
 TotaMiddleware totam;
 // agent constructor
 public FlockingAgent(TotaMiddleware totam) {
  this.totam = totam;
  }
 // agent body
 public void start() {
  totam.inject(new
  FlockingTuple("peerName1"));
  totam.inject(new
QueryTuple("peerName2,peerName3,peerName4"));
totam.subscribe(new
Tuple("GradFlock","peerName2",null,"peerName"
,null,"peerName4",null),"move");
}
// reaction method
public void move(Tuple event) {
  ...
  determine the closest downhill peer, by
comparing values in the GradFlock tuple with the
flocking tuples' values stored locally. Then
move towards that direction exploiting the
coordinates in the GradFlock tuple.
  ...
  }
}
```

We did not show the complete agent code for brevity. However the idea is that as the peer moves in its environment it queries all its neighboring peers for the GradFlock tuples, they are able to produce. The query process is managed by TOTA, in that it tries to maintain the consistency of the one-hop propagation rule for the query tuple. As peers answer, the enquiring peer can compute the gradients of all the flocking tuples and decide to move towards the downhill direction.

6. Related Works

A number of recent proposal address the problem of defining supporting environments for the development of adaptive, dynamic, context-aware distributed applications, suitable for pervasive computing.

Lime [PicMR01] exploits transiently tuple spaces as the basis for interaction in dynamic network scenario. Each mobile device, as well as each network nodes, owns a private tuple space. Upon connection with other devices or with network nodes, the privately owned tuple spaces can merge in a federated tuple space, to be used as a common data space to exchange information. TOTA subsumes and extend the Lime model. It is possible, via specific propagation rules, to have tuples distributed only in a local neighborhood, so as to achieve the same functionalities of a locally shared tuple space. In addition, propagation rules enable much more elaborated kinds of information sharing other than simple local merging of information. Similar considerations may apply with regard to other proposals for shared distributed data structures (e.g., the XMIDDLE trees [MasCE01]), which can be easily realized via tuple with proper content and propagation rules.

Smart Messages [Bor02], rooted in the area of active messages [Wet99], is an architecture for computation and communication in large networks of embedded systems. Communication is realized by sending "smart messages" in the network, i.e., messages which include code to be executed at each hop in the network path. The execution of the message at each hop determines the next hop in the path, making messages responsible for their own routing. Smart Messages share with TOTA, the idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunk of code to determine their propagation. However, in Smart Messages (and active messages) code is used mainly for routing or mobility purposes. In TOTA instead, the tuples are not simply routed though the network, but can are persistent and create a distributed data structure that remains stored in the network, and TOTA propagation code can also be used to change the message/tuple content, in order to realize distributed data structure and not just replicas.

The approach proposed in [RomJH02] to gather contextual information in a MANET scenario is quite similar to the one we exploit in TOTA for the same purpose. There, each peer in the MANET dynamically builds via propagation a distributed data structure (i.e., a shortest path tree) to gather contextual, location-sensitive, information from other peers in the network (also exploiting spatial localization information, as we have described in Subsection 2.3). TOTA is much more flexible: the possibility of programming propagation rules makes it possible to express coordination patterns and to drive navigation, other than just gather contextual information.

As a final note, we emphasize that *(i)* recent approaches in the area modular robots [KubCH01] exploit the idea of propagating "coordination field" across the robot agents so as to achieve a globally coherent behavior in robot's re-shaping activities; *(ii)* in the popular simulation game "The Sims" [Sims], characters move and act accordingly to specific fields that are assumed to be spread in the simulated environment and sensed by characters depending on situations (e.g., they sense the food field when hungry); *(iii)* ant-based optimization systems [Bon99, ParBS02] exploit a virtual environment in which ants can spread pheromones, diffusing and evaporating in the environment according to specific rules. *(iv)* amorphous computers [But02, Nag02] exploit propagation of fields to let particles self-organize their activities. Although serving different purposes, these approaches definitely share with TOTA the same physical inspiration.

7. Conclusions and Future Works

Tuples On The Air (TOTA) promotes programming distributed pervasive applications by relying on distributed data structures, spread over a network as sorts of electromagnetic fields, and to be used by application agents both to extract contextual information and to coordinate with each other in an effective way. As we have tried to show in this paper, TOTA suits the needs of pervasive computing environments (spontaneous interoperation, effective context-awareness, lightness of supporting environment) and facilitates both the access to distributed information and the expression of distributed coordination patterns.

Several issues are still to be investigated to make TOTA a practically useful framework for the development of pervasive applications. First, proper access control models must be defined to rule accesses to distributed tuples and their updates (and this is indeed a challenging issue for the whole area of pervasive and mobile computing). Second, we think it would be necessary to enrich TOTA with mechanisms to compose tuples with each other, so as to enable the expression of unified distributed data structures from the emanation of multiple sources, as it may be needed for scalability purposes. Finally, deployment of TOTA applications in real-world scenarios will definitely help identify current shortcomings and directions of improvement.

References

[BelPR01] F. Bellifemine, A. Poggi, G. Rimassa, "JADE - A FIPA2000 Compliant Agent Development Environment", 5th International Conference on Autonomous Agents (Agents 2001), Montreal (CA), May 2001.

[Bon99] E. Bonabeau, M. Dorigo, G. Theraulaz, "Swarm Intelligence", Oxford University Press, 1999.

[Bor02] C. Borcea, et al., "Cooperative Computing for Distributed Embedded Systems", 22th International Conference on Distributed Computing Systems, Vienna (A), IEEE CS Press July 2002.

[But02] W. Butera, "Programming a Paintable Computer", PhD Thesis, MIT Media Lab, Feb. 2002.

[CabLZ02] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", IEEE Transactions on Software Engineering, 20(10), Oct. 2002.

[CugFD01] G. Cugola, A. Fuggetta, E. De Nitto, "The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS", IEEE Transactions on Software Engineering, 27(9): 827-850, Sept. 2001.

[EstC02] D. Estrin, D. Culler, K. Pister, G. Sukjatme, "Connecting the Physical World with Pervasive Networks", IEEE Pervasive Computing, 1(1):59-69, Jan. 2002.

[FreHA99] E. Freeman, S. Hupfer, K. Arnold, "JavaSpaces Principles, Patterns, and Practice", Addison-Wesley, 1999.

[GelC92] D. Gelernter, N.Carriero "Coordination Languages and Their Significance", Communication of the ACM, 35(2):96-107, Feb. 1992.

[HigB01] J. Hightower, G. Borriello, "Location Systems for Ubiquitous Computing", IEEE Computer, 34(8):57-66, Aug. 2001.

[Jini] http://www.jini.org

[KubCH01] J. Kubica, A. Casal, T. Hogg, "Agent-based Control for Object Manipulation with Modular Self-Reconfigurable Robots", International Joint Conference on Artificial Intelligence, Seattle (WA), Aug. 2001, pp. 1344-1352.

[MamZL02] M. Mamei, F. Zambonelli, L. Leopardi, "Tuples on the Air: a Middleware for Context-aware Pervasive Computing", Technical Report No. DISMI-2002-23, University of Modena and Reggio Emilia, August 2002.

[MamLZ02] M. Mamei, L. Leonardi, M. Mahan, F. Zambonelli, "Coordinating Mobility in a Ubiquitous Computing Scenario with Co-Fields", Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices, AAMAS 2002, Bologna, Italy, July 2002.

[MasCE01] C. Mascolo, L. Capra, W. Emmerich, "An XML based Middleware for Peer-to-Peer Computing", 1st IEEE International Conference of Peer-to-Peer Computing, Linkoping (S), Aug. 2001.

[Mts2002] International Workshop on Mobile Teamwork Support, http://www.infosys.tuwien.ac.at/motion/mts/

[Nag02] R. Nagpal, "Programmable Self-Assembly Using Biologically-Inspired Multiagent Control", 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna (I), ACM Press, July 2002.

[ParBS02] V. Parunak, S. Bruekner, J. Sauter, "ERIM's Approach to Fine-Grained Agents", NASA/JPL Workshop on Radical Agent Concepts, Greenbelt (MD), Jan. 2002.

[PicMR00] G. P. Picco, A.M. Murphy, G.-C. Roman, "Software Engineering for Mobility: A Roadmap", in The Future of Software Engineering, A. Finkelstein (Ed.), ACM Press, pp. 241-258, 2000.

[PicMR01] G. P. Picco, A. L. Murphy, G. C. Roman, "LIME: a Middleware for Logical and Physical Mobility", In Proceedings of the 21st International Conference on Distributed Computing Systems, IEEE CS Press, July 2001.

[Rat01] S. Ratsanamy,, P. Francis, M. Handley, R. Karp, "A Scalable Content-Addressable Network", ACM SIGCOMM Conference 2001, San Diego (CA), ACM Press, Aug. 2001.

[RomJH02] G.C. Roman, C. Julien, Q. Huang, "Network Abstractions for Context-Aware Mobile Computing", 24th International Conference on Software Engineering, Orlando (FL), ACM Press, May 2002.

[RowD01] A. Rowstron, P. Druschel, "Pastry: Scalable,

Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems", 18th IFIP/ACM Conference on Distributed Systems Platforms, Heidelberg (D), Nov. 2001.

[Sims] The Sims, www.thesims.com

[Wet99] D. Wetherall, "Active network vision and reality: lessons from a capsule-based system", 17th ACM Symposium on Operating Systems Principles (SOSP '99) Published as Operating Systems Review 34(5):64–79, Dec. 1999.