Programming Pervasive and Mobile Computing Applications with the TOTA Middleware

Marco Mamei, Franco Zambonelli

DISMI – Università di Modena e Reggio Emilia – Via Allegri 13 – Reggio Emilia – ITALY
{ mamei.marco, franco.zambonelli }@unimo.it

Abstract

Pervasive computing calls for suitable middleware and programming models to deal with large software systems dived in dynamic mobile network environments. Here we present the programming model of TOTA ("Tuples On The Air"), a novel middleware for supporting adaptive context-aware activities in pervasive computing scenarios. The key idea in TOTA is to rely on spatially distributed tuples, propagated across a network on the basis of application-specific rules, for both representing contextual information and supporting uncoupled interactions between application components. As shown with the help of a case study scenario, TOTA promotes a simple programming model and can effectively facilitate access to distributed information, navigation in complex networks, and achievement of complex coordination tasks in a fully distributed and adaptive way.

1. Introduction

Computing is becoming intrinsically pervasive and mobile [15]. 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 [13], to monitor and control our environments [3], and to improve our interactions with the physical world [9]. Also, since most of the embeddings will be intrinsically mobile, 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 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 a flexible 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* possibly requiring light supporting infrastructures.

Unfortunately, current practice in distributed software development, as supported by currently available middleware infrastructures, is unlikely to effectively address the above requirement: *(i)* application agents are typically strictly coupled in their interactions (e.g., as in message-passing models and middleware), 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 builds on the lessons of uncoupled coordination models like event-based [5] and tuple space programming [4] 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. Specifically, in the TOTA ("Tuples On The Air") middleware, all interactions between agents take place in a fully uncoupled way via tuple exchanges. However, there is not any notion like a centralized shared tuple space. Rather, tuples can be "injected" into the network from any node and can propagate and diffuse accordingly to tuple-specific propagation patterns. The middleware takes care of propagating the tuples and of adapting their shape

accordingly to the dynamic changes that can occur in the network (as due by, e.g., mobile or ephemeral nodes). Agents can exploit a simple API to define and inject new tuples in the network and to locally sense both tuples and events associated with changes in the tuples' distributed structures (e.g., arrival and dismissing of tuples).

2. Motivations and Case Study

To sketch the main motivations behind TOTA, we introduce 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 a variety of tourists moving within it. We assume that each of them 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 with rooms, corridors, 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 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). In the following, we will concentrate on two specific representative problems: *(i)* how tourists can gather and exploit information related to an art piece they want to see; *(ii)* how tourists can meet in 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 move in the museum. They are likely to come and go at any time. Art pieces can be moved around the museum during special exhibitions or 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 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 supporting 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 [9], mobile robots and sensor networks [15]. Therefore, also all our considerations are of a more general validity, besides the addressed case study.

2.2. Inadequacy of Traditional Approaches

Most coordination models and middleware used so far in the development of distributed applications appear inadequate in supporting coordination activities in pervasive computing scenarios.

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 [7], as well as FIPA agent-based systems [1], support such a direct communication model. One problem of this approach is that agents, by having to interact directly with each other, can hardly sustain 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 contextual abstraction. In the case study scenario, tourists have to explicitly discover the location of art pieces, or of other tourists. Also, to orchestrate their movements, tourist 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 end 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 EventHeap [8], or they can be fully distributed over the nodes of the network, as in MARS [4]. In these cases, agents are no longer strictly coupled in their interactions, because tuple spaces mediate interactions and promote 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. Tourists can easily discover what art pieces are nearby them, but to locate a farther 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 to the target. Similarly, tourists can build an internal representation of the other people distribution 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: Siena[5] and Jini Distributed Events [7]. 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 tourist notify its movements across the building to the rest of the group. Notified agents can then easily obtain an updated picture of the current group distribution in a simpler and less expensive way than required by adopting shared data spaces. However, this approach still relies on agents for negotiating the coordinated movements and does not alleviate their computational tasks (i.e., in the case study, tourists 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 [4, 8] and event-based ones [5, 7] and extends them to provide agents with a unified and flexible mechanism to deal with both context representation and components' interaction.

In TOTA, we propose relying on distributed tuples for both representing contextual information and enabling uncoupled interaction among distributed application components. 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. Thus, TOTA tuples form a sort of spatially distributed data structure able to express not only messages to be transmitted between application components but, more generally, some contextual information on 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 neighbor 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 nodes are found within the range of their wireless connection.

Upon the distributed space identified by the dynamic network of TOTA nodes, each component is capable of locally storing tuples and letting them diffuse through the network. Tuples are injected in the system from a particular node, and spread hop-by-hop accordingly to their propagation rule. In fact, a TOTA tuple is defined in terms of a "content", and a "propagation rule". $T=(C,P)$. The content C is an ordered set of typed fields representing the information carried on by the tuple. The propagation rule P determines how the tuple should be distributed and propagated across the 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. In fact, tuples are not necessarily distributed replicas: by assuming different values in different nodes, tuples can be effectively used to build a distributed overlay data structure expressing some kind of contextual and spatial information (see figure 1). So, unlike event based models, propagation of tuples is not driven by a publish-subscribe schema, but it is directly encoded in tuples' propagation rule and, unlike an event, can change its content during propagation.

The spatial structures induced by tuples propagation must be maintained coherent despite network dynamism (see figure 1). To this end, 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 nodes 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 nodes. Similarly, when the topology changes due to nodes' movements, the distributed tuple structure automatically changes to reflect the new topology. For instance, figure 1 shows how the structure of a distributed tuple can be kept coherent by TOTA in a MANET scenario, despite dynamic network reconfigurations.

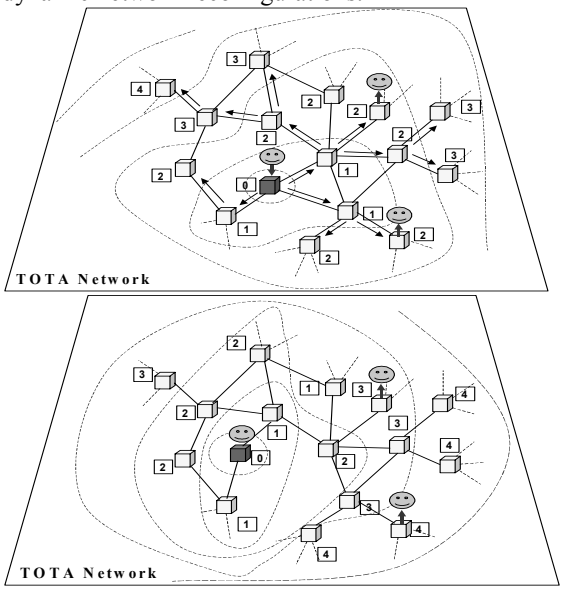

Figure 1: **(Top) the general scenario of TOTA: application components live in an environment in which they can inject tuples that autonomously propagate and sense tuples present in their local neighborhood. The environment is realized by means of a peer-to-peer network in which tuples propagates by means of a multi-hop mechanism. (Bottom) when the tuple source moves, tuples are updated to take into account the new topology**

From the application components' point of view, executing and interacting basically reduces to inject tuples, perceive local tuples and local events, and act accordingly to some application-specific policy. Software components 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 either the local tuple space or the one-hop neighbor tuple spaces to check for the presence of specific tuples. In addition, components can be notified of events (e.g., changes in tuple space content) occurring either locally or in the one-hop neighborhood.

3.1. 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 tourists 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. Moreover, we make the following assumptions: *(i)* devices are provided with localization mechanisms enabling them to know neighbors' coordinates in a private local coordinate frame. *(ii)* The rough topology of the ad-hoc network being formed by TOTA devices resembles the museum floor-plan. This means that there are not network links between physical barriers (like walls). To achieve this property, we suppose that either the devices are able to detect and drop those network links crossing physical barriers (e.g. relying on signal strength attenuation or some other sensors installed on the device) or that the museum building is pre-installed with a network backbone – reflecting its floor-plan – and that all the nodes can connect only to the backbone. And there are not long-range, wired backbones in the network. To achieve this property it is possible to rely on the natural physical attenuation of radio-based signals (in wireless communication), or to constrain the addressable space of wired nodes, to let them able to talk only with close peers.

Coming back to the case study, the first problem we face is that of enabling a tourist to discover the presence and the location of a specific art piece. TOTA makes this very simple, we could consider that art pieces can sense the income of tuples propagated by tourists – and describing the art piece they are looking for – and are programmed to react to these events by propagating backward to the requesting tourists a tuple containing their own location information. In particular, *Query and Answer tuples* could be defined as described in figure 2. Since TOTA keeps the tuple shape coherent despite node movements, *Query tuples* create gradients leading to their sources even if the sources move. Thus *Answer tuples* can reach a tourist while he/she is in movement.

The second problem we consider involves a "meeting" service whose aim is to help a group of tourists to find and move towards the most suitable room for a meeting. Even if several different policies can be though related to how a group of tourists should meet, here we will concentrate on having a group of tourists that wants to meet in the room that is between them (their "barycenter"). To this purpose, each tourist involved in the meeting can inject the *Meeting tuple* described in figure 2. Then, any tourist can follow downhill the tuple propagated by the farther other tourist in the group. In this way all tourists "fall" towards each other, and they meet in their barycenter room. It is interesting to notice,

that this room is evaluated dynamically and the process takes into account unexpected situations (e.g. crowded areas). So if some tourists encounter crowd in their path to the meeting room, the meeting room is automatically changed to a room closer to these unlucky tourists.

Query tuple
C= (description , distance)
P=(propagate to all peers hop by hop, increasing the "distance" field by one at every hop)
Answer tuple
C = (description, location, distance)
P = (propagate following downhill the "distance" of the associated query tuple, incrementing distance value by one at every hop)
Meeting tuple
C= (tourist_name, distance)
P=(propagate to all peers hop by hop, increasing the "distance" field by one at every hop)

Figure 2: **General description of the tuples involved in the case study scenario: Query Tuple, Answer Tuple and Meeting Tuple**

3.2. Implementation

From an implementation point of view, we developed a first prototype of TOTA running on Compaq IPAQs, running Linux (Familiar distribution) and equipped with 802.11b and Java 2 Micro Edition (J2ME, CDC, Personal profile). IPAQs connect locally in the MANET mode (i.e. without requiring access points) creating the skeleton of the TOTA network. Moreover, we have implemented a simulator to analyze TOTA behavior in presence of hundreds of nodes. The simulator, developed in Java, enables examining TOTA behavior in a MANET scenario, in which nodes topology can be rearranged dynamically either by a drag and drop user interface or by autonomous nodes' movements. The strength of our simulator is that, by adopting well-defined interfaces between the simulator and the application layers, the *same* code "installed" on the emulated devices can be installed on real devices. This allow to test applications first in the simulator, then to upload them directly in a network of real devices. Further details on the implementation can be found in [10].

4. TOTA Programming

When developing applications upon TOTA, one has basically to know: *(i)* what are the primitive operations to interact with the middleware; *(ii)* how to specify tuples and their propagation rule; *(iii)* how to exploit the above to code agents.

```
public void inject (TotaTuple tuple);
public Vector read (Tuple template);
public Vector readOneHop (Tuple template);
public Tuple keyrd (Tuple template);
public Vector keyrdOneHop(Tuple template);
public Vector delete (Tuple template);
public void subscribe (Tuple template,
ReactiveComponent comp, String rct);
public void unsubscribe (Tuple template,
ReactiveComponent comp);
```

Figure 3: **TOTA API**

```
public class ToyAgent implements AgentInterface
{
 private TotaMiddleware tota;
 /* agent body */
 public void start() {
  /* create a tuple and inject it*/
  FooTuple foo = new FooTuple("Hello World!");
  tota.inject(foo);
  /* define a template tuple */
  FooTemplTuple t = new FooTempTuple();
  /* read local tuples matching the template */
  Vector v = tota.read(t);
  /* subscribe to changes in tuples matching t*/
  tota.subscribe(t,this,"");
 }
 /* code of the reaction to the subscrption */
 public void react(String reaction, String
event)  { System.out.pritnln(event);}}
```

Figure 4: **Example code of a ToyAgent accessing the TOTA API**

4.1. TOTA Primitives

TOTA is provided with a simple set of primitive operations to interact with the middleware (see figure 3). *inject* 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 *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 *readOneHop* primitive returns a collection of the tuples present in the tuple spaces of the node's one-hop neighborhood and matching the template tuple. Each TOTA distributed tuple is also marked with an unique *id* (invisible at the application level) enabling a fast access to the tuple, disregarding its content. The *keyrd* and *keyrdOneHop* methods serve to this purpose, they look for tuples with the same *id* of the tuple passed as argument. The typical usage of these methods is to evaluate how a specific tuple has changed in the neighborhood. Specifically in the case of tuples with a numeric content, it allows to evaluate the tuple's gradient. The *delete* primitive extracts from the local middleware all the tuples matching the template and returns them to

the invoking agent. In addition, *subscribe* and *unsubscribe* primitives are defined to handle events. 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 in response to the occurrence of events matching the template tuple passed as first parameter. Specifically, when the a matching event happens, the middleware invokes on the agent a special *react* method passing as parameters, the reaction string and the matching event. The unsubscribe primitives removes matching subscriptions

The simple application agent in figure 4 clarifies the above concepts.

4.2. Specifying Tuples

Relying on an object oriented methodology, TOTA tuples are actually objects: the object state models the tuple content, while the tuple's propagation has been encoded by means of a specific *propagate* method.

When a tuple is injected in the network, it receives a reference to the local instance of the TOTA middleware, then its code is actually executed (the middleware invokes the tuple's *propagate* method) and if during execution it invokes the middleware *move* method, the tuple is actually sent to all the one-hop neighbors, where it will be executed recursively. During migration, the object state (i.e. tuple content) is properly serialized to be preserved and rebuilt upon the arrival in the new host.

Following this schema, we have defined an **abstract class TotaTuple**, that provides a general framework for tuples programming (see figure 5).

```
abstract class TotaTuple {
protected TotaInterface tota;
/* the state is the tuple content */
…
/* this method inits the tuple, by giving a
reference to the current TOTA middleware */
public void init(TotaInterface tota) {
 this.tota = tota; }
/* this method codes the tuple actual actions */
public abstract void propagate();
/* this method enables the tuple to react to
happening events… see later in the article */
public void react(String reaction, String event)
 {}}
```

Figure 5: **The structure of the TotaTuple class**

It is worth noting that a tuple is not thread by its own, it is actually executed by the middleware, that runs the tuple's *init* and *propagate* methods. The point to understand is that when the middleware has finished the execution of the tuple's methods, the tuple (on that node) becomes a 'dead' data structure eventually stored in the middleware local tuple space.

Tuples, however, must remain active even after the middleware has run their code. This is fundamental because their maintenance algorithm – see Section 5 - must be executed whenever the right conditions appear (e.g. a new peer has been connected). To this end, tuples can place subscriptions, to the TOTA event engine as provided by the standard TOTA API. These subscriptions let the tuples remain 'alive', being able to execute upon triggering conditions.

This model for tuples gives the maximum flexibility. However, the problem is that it is too complex, and we do not foster the idea of having the programmer to write tuples code form scratch. For this reason, we have developed a tuples' class hierarchy from which the programmer can inherit to create custom tuples without worrying about most of all the intricacies of dealing with tuple propagation and maintenance.

The only child of the *TotaTuple* class, is the **class StructureTuple**. This class is a template to create distributed data structures over the network. However, *StructureTuples* are NOT maintained by the middleware. This means that if the topology of the network changes the tuple local values are left untouched. This class inherits from *TotaTuple* and implements the superclass method propagate (see figure 6).

```
public final void propagate() {
 if(decideEnter()) {
  boolean prop = decidePropagate();
  changeTupleContent();
  this.makeSubscriptions();
  tota.store(this);
  if(prop)  tota.move(this); }}
```

Figure 6: **Standard implementation of the propagate method in the StructureTuple class**

The class *StructureTuple* implements the methods: *decideEnter*, *decidePropagate*, *changeTupleContent* and *makeSubscriptions* so as to realize a breadth first, expanding ring propagation. The result is simply a tuple that floods the network without changing its content.

Specifically, when a tuple arrives in a node (either because it has been injected or it has been sent from a neighbor node) the middleware executes the *decideEnter* method that returns true if the tuple can enter the middleware and actually execute there, false otherwise. The standard implementation returns true if the middleware does not already contain that tuple.

If the tuple is allowed to enter the method *decidePropagate* is run. It returns true if the tuple has to be further propagated, false otherwise. The standard implementation of this method returns always true, realizing a tuple's that floods the network being recursively propagated to all the peers.

The method *changeTupleContent* change the content of the tuple. The standard implementation of this method does not change the tuple content.

The method *makeSubscriptions* allows the tuple to place subscriptions in the TOTA middleware. In this way the tuple can react to events even when they happen after the tuple completes its execution. The standard implementation does not subscribe to anything.

After that, the tuple is inserted in the TOTA tuple space by executing *tota.store(this)*. Then, if the decidePropagate method returned true, the tuple is propagated to all the neighbors via the command *tota.move(this)*. Note that these last two commands are not in the TOTA API, since their access is restricted to tuples only.

The tuple will eventually reach neighboring nodes, where it will be executed again. It is worth noting that the tuple will arrive in the neighboring nodes with the content changed by the last run of the *changeTupleContent* method.

Programming a TOTA tuple to create a distributed data structure basically reduces at inheriting from the above class and overloading the four above methods to customize the tuple behavior. Here in the following, we present two examples to show the expressiveness of the introduced framework.

A *NMGradient* tuple creates a tuple that floods the network in a breadth-first way and have an integer hop-counter that is incremented by one at every hop (see figure 7). To code this tuple one has basically to:

- place the integer hop counter in the object state
- overload *changeTupleContent*, to let the tuple change the hop counter at every propagation step
- overload *decideEnter* so as to allow the entrance not only if in the node there is not the tuple yet – as in the base implementation -, but also if there is the tuple with an higher hop-counter. This allows to enforce the breadth-first propagation assuring that the hop-counter truly reflects the hop distance from the source.

A ***DownhillTuple*** creates a tuple that follows another *NMGradientTuple* downhill (see figure 8). To code this tuple one has basically to:

- overload the *decideEnter* method to let the tuple enter only if the value of the *NMGradientTuple* in the node is less that the value on the node from which the tuple comes from.

The rest of the hierarchy has been built in the same way: by overloading the methods controlling tuple propagation. Programmers can inherit from the hierarchy to further customize their tuple's propagation. The only point they have to remember is to call the superclass implementation before actually writing their own overload, to be sure that code we developed in the hierarchy is actually executed. In the following we give a brief overview of the rest of the hierarchy.

```
public class NMGradient extends StructureTuple {
 public int hop = 0;

public boolean decideEnter() {
 super.decideEnter();
 NMGradient prev =(NMGradient)tota.keyrd(this);
 return (prev == null ||
        prev.hop > (this.hop + 1));
}
protected void changeTupleContent() {
 super.changeTupleContent();
 hop++;}}
```

Figure 7: **Tuple example: NMGradient class**

```
public class DownhillTuple extends
StructureTuple {
 public int oldVal = 9999;
 NMGradientTuple trail;

 public DownhillTuple(String toFollow) {
  trail = new NMGradientTuple();
  trail.setContent(toFollow);}

 public boolean decideEnter() {
  super.decideEnter();
  int val = getGradientValue();
  if(val < oldVal) {
   oldVal = val;
   return true;
  }
  else
   return false;}

 /* this method returns the minimum hop-value of
the NMGradient tuples matching the tuple to be
followed in the current node */
 private int getGradientValue() {
  Vector v = tota.read(trail);
  int min = 9999;
  for(int i=0; i<v.size(); i++) {
   NMGradientTuple gt =
   (NMGradientTuple)v.get(i);
   if(min > gt.hop)
    min = gt.hop;
  }
 return min;}}
```

Figure 8: **Tuple example: DownhillTuple class**

MessageTuples are used to create messages that are not stored in the local tuple spaces, but just flow in the network. The basic structure is the same as StructureTuple, but a default subscription is in charge to erase the tuple after some time passed.

HopTuples create distributed data structure that are maintained by the TOTA middleware, to reflect changes in the network topology. Basically this class overloads the empty *makeSubscriptions* method of the *StructureTuple* class, to let these tuples react to changes

in the topology, by adjusting their values to always be consistent with the hop-distance form the source. It is worth noting that if the *NMGradient* tuple would have been inherited from *HopTuples* the resulting tuple would have been adaptive to source movements.

MetricTuples and ***SpaceTuples*** rely on spatial distances rather than hop-distances. The assumption here is to have radar-like location devices installed on nodes able to spatially localize neighboring (directly accessible) nodes. In other words, each device must be able to create a local private coordinate system by localizing neighborhood nodes. The basic implementation of these tuples, from which to inherit, is a tuple that combines local coordinate systems, to create a shared coordinate system, with the center in the node that injected the tuple.

A detailed explanation of the whole class hierarchy is outside the scope of this paper. More detailed information can be found in [11].

4.3. Programming the Case Study

It is rather easy now to program the agents required in our case study. The tuples they will use are the *NMGradient* and the *DownhillTuple* presented in the previous section (actually, the very first line of *NMGradient* should be changed to inherit from *HopTuple*).

With regard to the problem of gathering contextual information. We consider that art pieces (represented by *ArtAgen*, see figure 9) are programmed in order to sense the income of query tuples propagated by tourists (represented by *QueryAgent,* see figure 10) and to react by propagating backward to the requesting tourists their location information.

More in detail, the *QueryAgent* performs just two simple operations: it injects in the network a tuple of class *NMGradient*. Then it subscribes to the income of all the *DownhillTuples* (which are assumed to describe an art piece and its location) having as the first field "*Monna Lisa*", the agent associated to the Monna Lisa painting is expected to generate. The associated reaction displayReaction is executed on receipt of such tuple to print out the content of the received event tuple.

Correspondently, each *ArtAgent* 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 a *DownhillTuple* that simply follows backward the query tuple to reach the tourist agent issuing the request.

With regard to the meeting application. The algorithm followed by meeting agents (see figure 11) is very simple: agents have to determine the farthest peer, and then move by following downhill that peer's presence tuple. In this way agents will meet in their barycenter.

```
public class ArtAgent implements AgentInterface
{
 private TotaMiddleware tota;
 /* piece of art description and location */
 private String description, location;
 /* agent body */
 public void start() {
 /* subscribe to the query */
 NMGradient query = new NMGradient();
 query.setContent(description);
 tota.subscribe(query,this,"answerQuery");
 }
 /*the reaction injects the answer tuple. The
   answer will be coded by a DownhillTuple
   following the query. The query is here
   referenced as OneHopIncTuple event */
 public void react(String reaction, String
event) {
  NMGradient query = Tuple.deserialize(event);
  DownhillTuple answer = new
DownhillTuple(query.content);
  answer.setContent(description+" "+location);
  tota.inject(answer); }}
```

Figure 9: **Agent example: ArtAgent**

```
public class QueryAgent implements
AgentInterface {
 private TotaMiddleware tota;
 /* agent body */
 public void start() {
  /* inject the query */
  NMGradient query = new NMGradient();
  query.setContent("Monna Lisa");
  tota.inject(query);
  /* subscribe to the answer: the answer will be
   conveyed in a DownhillTuple, see 6.1 */
  DownhillTuple answer = new DownhillTuple();
  answer.setContent("Monna Lisa *");
  tota.subscribe(answer,this,"display"); }
 /* the reaction simply prints out the result */
 public void react(String reaction, String
event) {
  if(reaction.equalsIgnoreCase("display ")) {
  System.out.pritnln("Monna Lisa:" + event); }}
```

Figure 10: **Agent example: QueryAgent**

```
public class MeetingAgent extends Thread
implements AgentInterface {
 private TotaMiddleware tota;
 …
 public void run() {
// inject meeting tuple to participate meeting
  NMGradient mt = new NMGradient();
  mt.setContent(peer.toString());
  tota.inject(mt);
  while(true) {
   /* read other agents' meeting tuples */
   NMGradient coordinates = new NMGradient();
   Vector v = tota.read(coordinates);
   /* evaluate the gradients and select the peer
   to which the gradient goes downhill */
   GenPoint destination = getDestination(v);
   /* move downhill following meeting tuple */
    peer.move(destination); } }…}
```

Figure 11: **Agent example: MeetingAgent**

5. Performances and Experiments

One of the biggest concerns regarding our model is about scalability and performances. How much burden is requested to the system to maintain tuples?

Due to page limit, we will concentrate in this section to HopTuples only, since they are the ones actually used in the paper's case study and are the most difficult to be maintained. Further details on these topics can be found in [11]. HopTuples' maintenance operations are required upon a change in the network topology, to have the distributed tuples reflect the new network structure. This means that maintenance operations are possibly triggered whenever, due to nodes' mobility or failures, new links in the network are created of removed. Because of scalability issues, it is fundamental that the tuples' maintenance operations are confined to an area neighboring the place in which the topology had changed. This means that, if for example, a device in a MANET breaks down (causing a change in the network topology) only neighboring devices should change their tuples' values. The size of this neighborhood is not fixed and cannot be predicted a-priori, since it depends on the network topology. For example, if the source of a tuple gets disconnected from the rest of the network, the updates must inevitably involve all the other peers in the network (that must erase that tuple form their repositories, see figure 12-top). However, especially for dense networks, this is unlikely to happen, and usually there will be alternative paths keeping up the tuple shape (see figure 12-bottom).

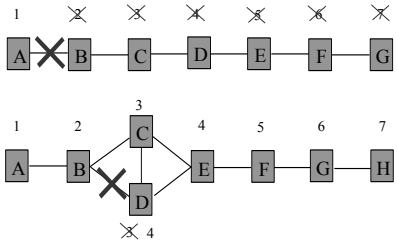

Figure 12: **The size of the update neighborhood depends on the network topology. Here is an example with a tuple incrementing its integer content by one, at every hop. (top) the specific topology force update operations on the whole network (bottom) if alternative paths can be found, updates can be much more localized.**

How can we perform such localized maintenance operations in a fully distributed way? To fix ideas, let us consider the case of a tuple incrementing its integer content by one, at every hop.

Given a local instance of such a tuple *X*, we will call *Y* a *X*'s *supporting tuple* if: *Y* belongs to the same distributed tuple as *X*, *Y* is one-hop distant from *X*, *Y* value is equal to *X* value minus 1.With such a definition, a *X*'s supporting tuple is a tuple that could have created *X* during its propagation.

Moreover, we will say that *X* is in a *safe-state* if it has a supporting tuple, or if it is the source of the distributed tuple. We will say that a tuple is not in a safe-state if the above condition does not apply.

Each local tuple can subscribe to the income or the removal of other tuples belonging to its same type in its one-hop neighborhood. This means, for example, that the tuple depicted in figure 12-bottom, installed on node F and having value 5 will be subscribed to the removal of tuples in its neighborhood (i.e. nodes E and G).

Upon a removal, each tuple reacts by checking if it is still in a safe-state. In the case a tuple is in a safe-state, the tuple the removal has not any effect - see later -. In the case a tuple is not in a safe state, it erases itself from the local tuple space. This eventually cause a cascading tuples' deletion until a safe-state tuple can be found, or the source is eventually reached, or all the tuples in that connected sub-network are deleted (as in the case of figure 12-top). When a safe-state tuple observes a deletion in its neighborhood it can fill that gap, and reacts by propagating to that node. This is what happens in figure 12-bottom, safe-state tuple installed on mode C and having value 3 propagates a tuple with value 4 to the hole left by tuple deletion (node D). It is worth noting that this mechanism is the same enforced when a new peer is connected to the network.

Similar considerations applies with regard to tuples' arrival: when a tuple sense the arrival of a tuple having value lower than its supporting tuple, it means that, because of nodes' mobility, a short-cut leading quicker to the source happened. Also in this case the tuple must update its value to take into account the new topology.

So, what is the impact of a local change in the network topology in real scenarios?

To answer these questions we exploited the implemented TOTA simulator, being able to derive results depicted in figure 13.

The graphs show results obtained by more than 100 experiments, conducted on different networks. We considered networks having an average density (i.e. average number of nodes directly connected to an other node) of 5.7, 7.2 and 8.8 respectively (these numbers come from the fact that in our experiments they correspond to 150, 200, 250 peers, respectively). In each network, a tuple, incrementing its content at every hop, had been propagated. Nodes in the network move randomly, continuously changing the network topology. The number of messages sent between peers to keep the tuple shape coherent had been recorded.

Figure 13-a shows the average number of messages

sent by peers located in an x-hop radius from the origin of the topology change. Figure 13-b shows the same values, but in these experiments only the source of the tuple moves, changing the topology. Figure 13-c shows the percentage of topology changes, happened during the experiments, that required a specific number of messages to be dealt with (see caption).

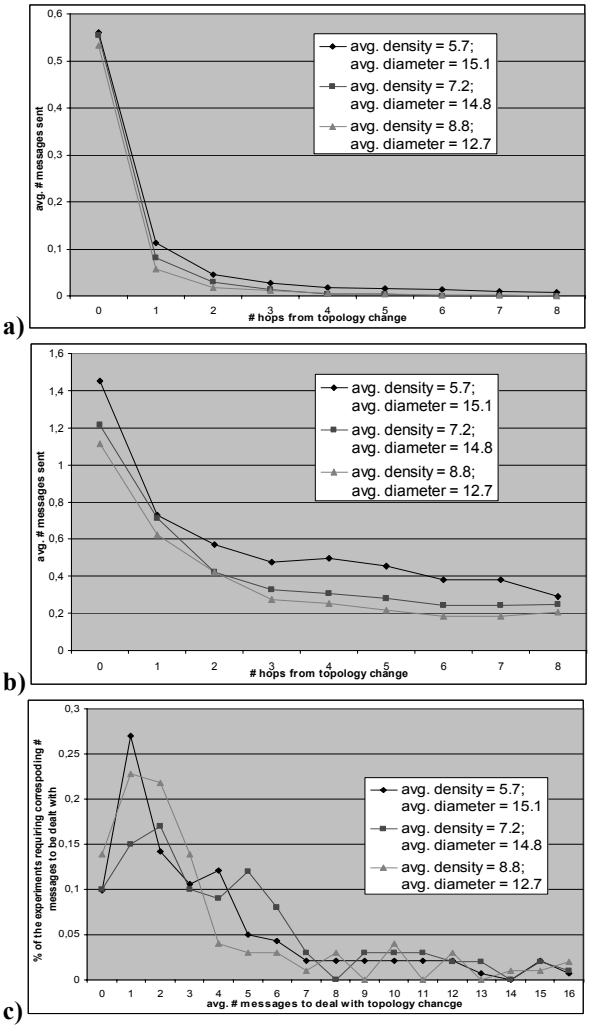

Figure 13: **Experimental results: locality scopes in tuple's maintenance operations emerge in a network without predefined boundaries. (a) topology changes are caused by random peer movements. (b) topology changes are caused by the movement of the source peer. (c) Number of topology changes, happened during the experiments, that required a specific number of messages to be dealt with (e.g. in 27% of the diamond experiment, the topology change has been fixed with about 1.5 messages being exchanged)**

The most important consideration we can make looking at those graphs, is that, upon a topology change, a lot of update operations will be required near the source of the topology change, while only few operations will be required far away from it. This implies that, even if the TOTA network and the tuples being propagated have no artificial boundaries, the operations to keep their shape consistent are strictly confined within a locality scope (figure 13-a-b).

Moreover, figure 13-c, shows that the topology change that are likely to involve large-scale update are much less frequent than operations requiring only local rearrangements. This fact supports the feasibility of the TOTA approach in terms of its scalability. In fact, this means that, even in a large network with a lot of nodes and tuples, we do not have to continuously flood the whole network with updates, eventually generated by changes in distant areas of the network. Updates are almost always confined within a locality scope from where they took place.

Other experiments, related to test the scalability of the system in other situations, are in our research agenda. For instance, it will be particularly interesting to see what happens when a large portion of the network topology changes, such as in networks of TOTA nodes embedded in vehicle or carried on by a person).

6. Related Works

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

Smart Messages (SM) [3], rooted in the area of active networks, 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. SM shares with TOTA, the general idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunk of code to determine their propagation. The main difference between SM and TOTA is that in SM messages tend to be used as light-weight mobile agents, roaming across the network, and performing different tasks. In TOTA tuples tend to form self-maintained distributed data structures guiding other agents in their task.

The L2imbo model, proposed in [6], is based on the notion of distributed tuple spaces augmented with processes (Bridging Agents) in charge of moving tuples form one space to another. Bridging agent can also change the content of the tuple being moved for example to provide format conversion between tuple spaces. The

main differences between L2imbo and TOTA are that in L2imbo, tuples are conceived as "separate" entities and their propagation is mainly performed to let them being accessible from multiple tuple spaces. In TOTA, tuples form distributed data structure and their "meaning" is in the whole data structure rather than in a single tuple. Because of this conceptual difference, tuples' propagation is defined for every single tuple in TOTA, while is defined for the whole tuple space in L2imbo.

Lime [14] 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 of Lime. 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 [12]).

7. Conclusions and Future Works

Several issues are still to be investigated to make TOTA a practically useful framework for the development of pervasive applications. In particular, a criticism that can apply to TOTA is the lack of an underlying general methodology, enabling engineers to map a specific coordination policy into the corresponding definition of tuples and of their shape. Personally, we believe that a great number of coordination patterns can be easily engineered in TOTA even in the absence of a general methodology (e.g., biological systems such as ant-colonies can be sources of several ready-to-work solutions [2]). Nevertheless, the definition of such a methodology – which is still lacking in all of the related approaches based on similar self-organization principles – would be definitely of help and would possibly make TOTA applicable to a wider class of distributed coordination problems.

Acknowledgments. Work supported by the Italian MIUR "Progetto Strategico IS-MANET, Infrastructures for Mobile ad-hoc Networks ".

References

[1] F. Bellifemine, A. Poggi, G. Rimassa, "JADE - A FIPA2000 Compliant Agent Development Environment", ACM Intl. Conference on Autonomous Agents, 2001.

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

[3] C. Borcea, et al., "Cooperative Computing for Distributed Embedded Systems", IEEE Intl. Conference Distributed Computing Systems, 2002.

[4] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", IEEE Transactions on Software Engineering, 28(11):371-380, Nov. 2002.

[5] A. Carzaniga, D. Rosenblum, A. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", ACM Transaction on Computer System, 19(3):332-383.

[6] N. Davies, et al, "L2imbo: A distributed systems platform for mobile computing", ACM Mobile Networks and Applications, 3(2):143-156.

[7] JINI, http://www.jini.org

[8] B. Johanson, A. Fox, "The Event Heap: A Coordination Infrastructure for Interactive Workspaces", IEEE Workshop on Mobile Computer Systems and Applications, 2002.

[9] M. Mamei, F. Zambonelli, L. Leonardi, "Co-Fields: A Physically Inspired Approach to Distributed Motion Coordination", IEEE Pervasive Computing, 2004, to appear.

[10] M. Mamei, F. Zambonelli, L. Leonardi, "Tuples On The Air: a Middleware for Context-Aware Computing in Dynamic Networks", IEEE Intl. Conference Distributed Computing Systems Workshops, 2003.

[11] M. Mamei, F. Zambonelli, "Self-Maintained Distributed Tuples for Field-based Coordination in Dynamic Networks", ACM Symposium on Applied Computing, Cyprus (CY), 2004.

[12] C. Mascolo, L. Capra, W. Emmerich, "An XML based Middleware for Peer-to-Peer Computing", IEEE Intl. Conference of Peer-to-Peer Computing, 2001.

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

[14] G. P. Picco, A. L. Murphy, G. C. Roman, "LIME: a Middleware for Logical and Physical Mobility", IEEE Intl. Conference Distributed Computing Systems, 2001.

[15] R. Want, G. Borriello, "Survey on Information Appliances", IEEE Computer Graphics and Applications, 20(3):24-31.