Programming Stigmergic Coordination with the TOTA Middleware

Marco Mamei, Franco Zambonelli
Dipartimento di Scienze e Metodi dell'Ingegneria
Università di Modena e Reggio Emilia
Via Allegri 13 – Reggio Emilia– ITALY
mamei.marco@unimore.it, franco.zambonelli@unimore.it

ABSTRACT

Stigmergic coordination has received a growing attention in the past few years. In fact, by decoupling interacting agents via the mediation of an active environment, stigmergy promotes the definition of robust and adaptive multiagent systems. However, beside a large amount of scientific studies, the problem of defining usable and general-purpose tools to program stigmergy-coordinated multiagent systems is still open. In this context, this paper shows how the TOTA middleware can be effectively exploited to support a variety of stigmergy-based coordination activities. The key idea in TOTA is to rely on a simple API for injecting tuple-based information in a network, have it propagate and/or evaporate accordingly to application-specific policies, and have it locally sensed by application agents. Application examples are presented to show that TOTA can promote a simple programming of a variety of different types of stigmergic interactions, in a variety of operational environments.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Multiagent Systems; C.2.4 [**Computer-Communication Systems**]: Distributed Systems;

General Terms

Algorithms, Design

1. INTRODUCTION

Stigmergy is being more and more recognized as a powerful approach to coordinate activities in complex multiagent systems [18]. Getting inspiration from nature, stigmergy has proved useful to enforce robust and self-adaptive situated behaviors in a variety of scenarios, ranging from P2P systems [2], robot swarms [14, 17] and sensor networks [8].

In general, stigmergy refers to all those kinds of indirect interactions occurring among situated agents that, by affecting and sensing the properties of a shared environment, reciprocally affect each others' behavior [9]. Such indirection makes stigmergic approaches intrinsically suited to large-scale and open multiagent systems, promoting self-organization. Also, by considering the possibility for an active environment to host internal processes affecting its own properties, stigmergy may support powerful forms of context-aware coordination, suitable to tackle the dynamics of modern scenarios and promoting self-adaptation. So far, the most widely used stigmergic mechanisms in multiagent systems include pheromone-based (getting inspiration from ant foraging, and relying on agents depositing markers that the environment can diffuse and evaporate) [18] and field-based ones (getting inspiration from physical force fields and relying on agents' being associated to a sort of "aura" that propagates in the environment) [12].

Despite the large number of applications, practical and usable tools for programming and supporting stigmergy coordinated applications are still missing. Besides scientific simulation studies, where such an issue is not of key relevance, those systems and applications deployed so far that exploit some forms of stigmergic coordination have always adopted specific ad-hoc programming and infrastructural solutions, without attempting at generalizing. However, as the interest in developing and deploying stigmergy coordinated applications increases, the need for general purpose and usable tools will soon become compulsory [22, 23].

The contribution of this paper is to show how the TOTA ("Tuples On The Air") middleware [10] can represent an effective answer to the above issue. The key ideas in TOTA are to:

- provide basic support for storing, propagation, and maintenance of distributed tuple-based data structures in dynamic networked environments;
- enable a simple application-level definition of tuples, of their propagation rules, and of their maintenance rules (e.g., evaporation rules);
- make available to agents a simple API for injecting tuples in a network (which then propagate and/or "evaporate" accordingly to the defined policies), and have them locally sensed by agents.

In this way – as shown via several application examples – TOTA enables developers and programmers to easily configure any specific type of stigmergic coordination – whether pheromone-based or field-based – and to have their deployment and execution properly supported.

This following of this paper is organized as follows. Section 2 discussed related work in the area. Section 3 introduces the TOTA middleware. Section 4 goes into details about the TOTA API and its programming model. Section 5 sketches two application examples. Section 6 concludes and discusses open issues.

2. RELATED WORK

In the past few years, a growing amount of research activities have been devoted to the study of stigmergic approaches – whether

AAMAS'05, July 2529, 2005, Utrecht, Netherlands.

pheromone-based [3] or field-based [12, 15] – for the coordination of complex multiagent systems.

Beside studies of a more scientific nature exploiting either analytical or simulation approaches [2, 3], several examples of deployed systems – in a variety of different scenarios and serving different applications – have been reported, e.g.: navigation in P2P networks [21], motion coordination in robot swarms [14, 17, 20], data gathering in sensor networks [8]; context-aware coordination in pervasive computing environments [12], just to mention a few examples. However, despite the recognition of the fact that stigmergic coordination may have wide applicability for a variety of distributed computing scenarios, no proposals so far explicitly focus on the identification and definition of proper general-purpose infrastructures and programming approaches.

The Amorphous Computing project [1], though, focuses on the somewhat close issue of identifying suitable models for programming applications over amorphous networks of "computing particles". To this end, a simple biologically-inspired programming language based on the propagation and local sensing of simple field-based data structures has been proposed [15]. The project so far has focused on very simple immobile particles, modeled as finite-state machines with a limited number of states – not much different from cellular automata cells. Thus, the effectiveness of the associated programming language to deal with systems of more complex, mobile, and situated particles (as agents in a situated MAS will be) is limited, calling for notable extensions [16]. Also, the focus is on field-based data structures, disregarding issues of e.g., pheromone evaporation.

Shifting to a totally different research area, recent researches on middleware infrastructures for pervasive and mobile computing are proposing a variety of novel coordination abstractions that get somewhat closer to stigmergic coordination than traditional message-passing and event-based middleware do.

In the Smart Messages systems [4], communications between agents/processes occur via sorts of "active" messages that can include code to be executed at each hop in the network path, so to dynamically modify the content of the message itself and/or its routing strategy. In other words, communication between agents involves the active mediation of an active environment, whose activities (encoded in active messages) can be properly configured. Although, it is potentially possible to program active messages to make them fully assimilated to pheromones and fields, this task requires notable programming efforts.

The L2imbo middleware [7] exploits distributed tuple spaces augmented with internal processes (Bridging Agents) that can move tuples around in the network form one space to another, and can also dynamically change their content. Also in this case, interactions are mediated by an active environment and, by properly defining tuples and bridging agents, one could think at reproducing forms of stigmergic coordination. However, also in this case, the required programming efforts would be notable.

Similar considerations apply to a variety of recently proposed coordination middleware relying on active/reactive components for dynamic data update and propagation, e.g. MARS [5], XMIDDLE [13], and LIME [19].

The TOTA middleware, described in the following, is explicitly conceived to meet the needs of stigmergic approaches to coordination, thus making it simple to define, program, and access both pheromones and fields, while preserving at the same time generality.

3. THE "TUPLES ON THE AIR" APPROACH

The TOTA middleware gathers concepts from both tuple space coordination architectures [5, 19] and event-based ones [6], but extends them so as to define a simple and flexible coordination mechanisms for general-purpose stigmergic coordination.

3.1 General Overview

In TOTA, we propose relying on distributed tuples for both representing contextual information and enabling stigmergic interactions among distributed agents.

Unlike traditional shared data space models for distributed computing, TOTA 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, diffuse, and evolve in the network accordingly to specified patterns. Thus, TOTA tuples form a sort of spatially distributed data structure able to express properties of the network environment that can be used to acquire contextual information about the environment itself, and to support the mechanisms via which stigmergic interactions can take place.

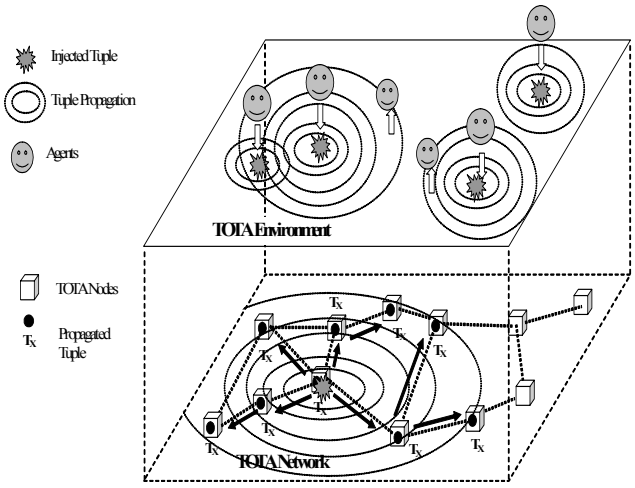


Figure 1: The TOTA scenario: a peer-to-peer network of nodes, each locally hosting the TOTA middleware and providing for tuple storing and propagation across the network. At the application level, agents perceive living in an environment in which they can indirectly interact by propagating and locally sensing tuples.

To support this idea, TOTA consider the presence of a peer-to-peer network of nodes, each node running a local version of the TOTA middleware (Figure 1). Each of these TOTA nodes holds references to a limited set of neighbor nodes. The structure of the network, as determined by the neighborhood relations, may be highly dynamic (due to node mobility, ephemeral nodes, or faults in nodes). TOTA assumes networking capability for recognizing connection and disconnection events.

The specific nature of the network scenario determines how each node can found its neighbors, and the overall logical structure of the TOTA network. In MANETs, sensor networks, and robot swarms scenarios, TOTA neighbor nodes are identified within the range of their wireless connection. In wired networks, TOTA

neighbors can be determined by some logical structure of the network or by some sort of overlay or social structure. More in general, independently of the network scenario, whenever the nodes of a network can be geographically localized, the structure of the TOTA network can also be based on such spatial information (for the sake of space limitations, we only mention the fact that TOTA can be itself used to enforce geographical self-localization of nodes in a network, and that tuples relying on such information are indeed available).

Upon the distributed space identified by the dynamic network of TOTA nodes, which can be considered as the environment in which agents situate, agents can execute on these nodes and exploit the API provided by TOTA to indirectly interact with each other, mostly disregarding network details, and simply injecting tuples from their local position in space and locally sensing propagated tuples.

3.2 TOTA Tuples

Unlike traditional "tuples" which simply have data content, a TOTA tuple is defined in terms of content, a propagation rule, and a maintenance rule:

$$T=(C,P,M)$$

The content *C* is an ordered set of typed fields representing the information carried on by the tuple. *C* can range from a simple integer value up to any structured data. In this way, TOTA tuples can embed not only simple pheromones and fields, but semantically richer properties of the environment.

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, if not isomorphic) 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 a tuple content should change while it is propagated. In fact, tuples are not necessarily distributed replicas, but can be effectively used to build a distributed overlay data structure expressing some kind of distributed properties of the network environment, e.g., pheromones and fields. For instance, to propagate a field expressing the network distance form the source, a propagation rule can simply propagate the tuples across all the network by increasing a numerical value at each network hop (Figure 2).

The maintenance rule *M* determines how a tuple's distributed structure should react to events occurring in the environment. These types of event can be simple time alarms, which are nevertheless of fundamental importance in pheromone-based coordination to enforce evaporation. Or they can be events associated to changes in the network structure, which is of fundamental importance to preserve a coherent structure of the environmental properties represented by fields. To this end, the TOTA middleware supports tuples propagation actively and adaptively: by constantly monitoring local events, the network local topology and the income of new tuples, the middleware automatically re-shape tuples and their distributed structure whenever appropriate w.r.t. the maintenance rule. 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. With reference to the "field" tuple example (Figure 2), the maintenance rule may require that the "field" continuously reflect the actual position of the source agent, whenever it moves. With regard to pheromones, the maintenance rule may require that a tuple is periodically locally updated, to enforce evaporation, and that it is periodically propagated, to enforce slow chemical diffusion.

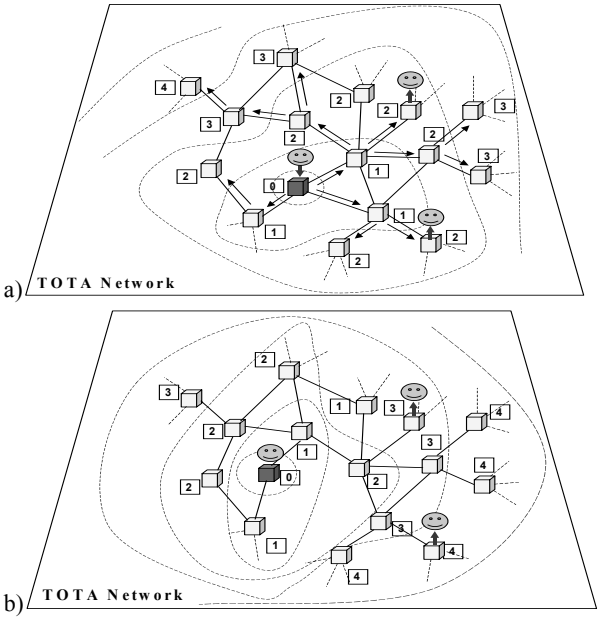


Figure 2: (a) An agent inject a "field" tuple increasing an integer value in its content as it propagates. Other agents can locally perceive the tuples and discover the distance from the source agent. (b) If the source agent (or its node) moves, the maintenance rule can specify to automatically update the distributed tuple structure to account for the new situation.

Generic Field Tuple

C = (field_type_identifier, distance)

P = (propagate everywhere, by incrementing distance by one at each network hop)

M = (update structure upon network topology changes)

Generic Pheromone Tuple

C = (pheromone_type_identified, strenght)

P = (propagate in neighborhood)

M = (evaporate by diminishing strength periodically)

Figure 3: Description of generic "field" and "pheromone" tuples, as to be realized in TOTA.

From the agents' viewpoint, executing and interacting basically reduces to inject tuples, perceive local tuples (as well as "tuple gradients", by accessing tuples in neighbor nodes) and local events, and act accordingly to some application-specific policy. Developers, by their side, are charged with the duty of identifying proper content, propagation, and maintenance rules for their application tuples (as, e.g., in Figure 3), and of properly coding them in tuple classes, as described in Section 4.

It is worth mentioning that TOTA, while conceived for stigmergic coordination, also subsumes more traditional forms of indirect interactions. Tuples with a null propagation rule, *T=(C, null, M)*

are simply local, as in reactive tuple space models [5]. Tuples with also a null maintenance rule *T=(C, null, null)*, promote traditional, non-reactive, tuple-based coordination models.

3.3 Implementation

From an implementation point of view, TOTA is fully developed in Java, thus it is highly portable. TOTA can actually run on any JVM-equipped computer device, and can run both in wired and wireless mode. Basically, the implementation includes a tuple space engine, to be locally accessed by agents via an API interface, an event-based engine to catch system-level and network-level events, and a reactive engine to execute the code for tuples propagation and maintenance when appropriate events occur.

We extensively experienced the described implementation of TOTA by using Compaq IPAQs, with Linux Familiar and 802.11b in ad-hoc mode, creating the skeleton of an ad-hoc TOTA network.

In addition to the actual implementation of TOTA, we have also implemented a simulator to analyze TOTA behavior in very large systems. The simulator enables examining TOTA behavior in any network scenarios. In addition, the simulator can execute in simulated nodes the *same* TOTA code of real devices, and enable "mapping" in a simulated scenario real network devices. This allow to test applications on a few real devices, while having them behave as if they were immersed in very large networks. See [10] for more details.

4 TOTA PROGRAMMING

Developing applications using the TOTA middleware basically implies knowing: *(i)* what are the primitive operations available in the API to interact with the environment; *(ii)* how to specify tuples, their propagation rules, and their maintenance rules; *(iii)* how to properly exploit the above in agents. This latter point will be the core of section 5.

4.1 TOTA Primitives

TOTA is provided with a simple set of primitive operations to interact with the middleware (see Figure 4). *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), and will be stored in each of the propagation nodes in accord to its maintenance rules. 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. In stigmergic coordination, such an operation is necessary to estimate "gradients" or either pheromones and fields. To make such an estimation more efficient, the *keyrd* and *keyrdOneHop* are also provided to access tuples based on their unique *id*. 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, system-level events) 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 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* primitive removes matching subscriptions.

The simple toy agent in Figure 5 clarifies the possible use of the TOTA primitives.

```
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 4: The 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 5: A ToyAgent exploiting the TOTA API.

4.2 Specifying Tuples

Being implemented in Java, TOTA tuples are actually objects: the object state models the tuple content, while the tuples' propagation and maintenance rules has been encoded by means of specific *propagate* and *react* methods, respectively.

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 a 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. The *abstract class TotaTuple* provides the basic class on which to rely to define – via inheritance – tuples to serve specific types of stigmergic coordination (Figure 6).

```
abstract class TotaTuple {
protected TotaInterface tota;

// the instance variables are 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 propagation rule
public abstract void propagate();

/* this method enables the tuple to react to
specific events, to perform maintenance */
public void react(String reaction, String event)
 {}
}
```

Figure 6: 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*, *propagate*, and *react* methods when necessary. 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 passive data structure stored in the middleware local tuple space. A tuple is re-activated whenever events for maintenance occur. It is up to tuples themselves to subscribe to interesting events.

While the proposed tuple models provides for the maximum flexibility, coding a specific tuple class from scratch starting from the class *TotaTuple* may be complex. For this reason, TOTA already provides a library of class hierarchies from which the programmer can inherit to create specific tuples without worrying about most of the low-level intricacies of dealing with tuple propagation and maintenance [11]. Of particular relevance is the *StuctureTuple* class. *StructureTuple* structures the *propagate* method of *TotaTuple* into four simple and well-defined sub-methods. Thus, subclassing from *StrucutreTuple* makes the writing of specific policies simpler, and amounting at overloading some of these simple sub-methods.

StructureTuple implements the *propagate* method accordingly to the schema depicted in Figure 7. The *decideEnter* methods is executed to assess whether the tuple can enter a specific node. The *makeSubscriptions* method allows the tuple to subscribe to relevant events to perform maintenance operations. The *changeTupleContent* method allows to change the tuple content to create not-dull data-structures. The *decidePropagate* method is executed to asses whether the tuple has to be further propagated to neighbor nodes or not. Further details on this schema in [11].

To make an example, Figure 8 shows the code of a *HopTuple* class (provided in the TOTA library) that implements a simple field expressing the network distance from the source (as in Figure 2). The *changeTupleContent* and *decidePropagate* methods are very simple. The *decideEnter* enforce a breadth first propagation, to avoid multiple propagations of the same tuple. The maintenance rule to re-shape the distributed field structure upon network topology changes is expressed in the *react* method, supported by the *makeSubscription* method ensuring that the maintenance rule is applied whenever appropriate. As exemplified in the following section, directly inheriting from *HopTuple* makes the writing of diverse types of fields extremely simple.

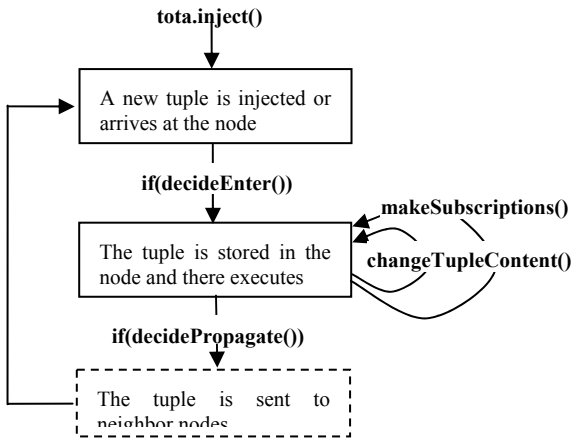


Figure 7: Standard template to create tuples by overloading *decideEnter*, *makeSubscriptions*, *changeTupleContent* and *decidePropagate* methods.

5 APPLICATION EXAMPLES

Let us not put the above tools at work in two exemplary case study applications.

5.1 Flocking with Fields

OVERVIEW: The goal of this application is to let a group of agents coordinate their movements to maintain a specific distance from each other while moving. To achieve this coordinated behavior, we take inspiration from a well-known example in swarm-intelligence [3]. Flocks of birds stay together, coordinate turns, and avoid each other, by following a very simple swarm algorithm [18]. Their coordinated behavior can be explained by assuming that each bird tries to maintain a specified separation from the nearest birds and to match their speeds velocity, so as to exploit (the same as cyclists do) useful aerodynamics effects.

To implement such behavior with TOTA, each agent can generate a tuple *FlockingTuple*, as a field whose value assumes its minimal value at the desired distance from the source, expressing the intended spatial separation between agents (these are network distances, measured in terms of network hops). The final shape of this field approaches the function depicted in Figure 9-a. *FlockingTuples* are always updated to reflect peers' movements. To coordinate movements, peers have simply to locally perceive the generated tuples and follow them downhill. The result is a globally coordinated movement, in which peers maintain an almost regular grid formation see Figure 9-b.

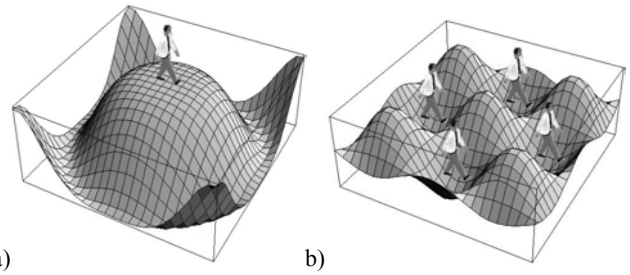
TUPLES: To code a *FlockingTuple* (Figure 10), one has to inherit from *HopTuple* and simply overload the method *changeTupleContent* so as to shape the counter propagation accordingly to Figure 9-a. Doing this is dramatically simple, and

preserves in *FlockingTuple* the proper maintenance rules to deal with dynamics.

AGENTS: Flocking agents are really simple. They inject flocking tuples then they follow flocking tuples downhill (Figure 11).

```
public class HopTuple extends StructureTuple
{ public int hop = 0; // initialize counter

protected void changeTupleContent() {
  hop++; // counter increased at each hop
 }

protected boolean decidePropagate() {
  return true; // propagates everywhere
}
// breadth first propagation
// enter a node only if not already there
 protected boolean decideEnter() {
  HopTuple prev = (HopTuple)tota.keyrd(this);
  return ((prev==null)||(prev.hop>(hop+1)));
 }

/* the tuple subscribes to any change in
the local structure of peers and to the
removal of instances of itself */
 protected void makeSubscriptions() {
  super.makeSubscriptions();
  PresenceTuple pres = new
        PresenceTuple("<peer=*>");
  TsTuple inPres =  new
      TsTuple("<op=IN><"+pres.serialize()+">");

  tota.subscribe(inPres, this,"PC");

   TsTuple tOut = new
     TsTuple("<op=OUT><"+this.serialize()+">");

   tota.subscribe(tOut, this,"OUT");
 }

 // react method to handle tuple maintenance
 public void react(String react, String event)
 {
  super.react(reaction,event);
  if(reaction.equalsIgnoreCase("PC"))
  {/* a tuple is in a "safe-state" if it is the
   one originally injected by the agent or if
   it has a neighbor tuple with a lower hop */
    if(safeState() && decidePropagate())
         tota.move(this);   }
    else if(reaction.equalsIgnoreCase("OUT"))
         { if(!safeState())
              tota.delete(this);
           else
              tota.move(this); }
 }
}
```

Figure 8: The code of the HopTuple class.



a) b)

Figure 9: Flocking overview. (a) Ideal shape of the flocking tuple. (b) When agents follow other agents' tuples, they self-organize in a regular grid formation.

```
public class FlockingTuple extends HopTuple {
 private int RANGE = 3;

public FlockingTuple(int RANGE) {
  this.RANGE = RANGE;
 }

 public int value = RANGE;
 protected void changeTupleContent() {
  super.changeTupleContent();
  if(hop <= RANGE)
     value --;
  else
     value ++;
  }
}
```

Figure 10: The FlockingTuple class.

```
public class FlockingAgent extends Thread
implements AgentInterface {

private TotaMiddleware tota;

public void run() {

/* create and inject the flocking tuple to
participate the flock */
FlockingTuple ft = new FockingTuple ();
ft.setContent(peer.toString());
tota.inject(ft);

while(true) {
  // read other agents' flocking tuples
  FlockingTuple query = new FlockingTuple();
  Vector v = tota.read(query);
  // select the peer where gradient goes downhill
  GenPoint destination = getDestination(v);
  // move downhill following the meeting tuple
  this.move(destination);}
 }
}
```

Figure 11: Agent example: FlockingAgent.

5.2 Routing with Pheromones

OVERVIEW: The goal of this application is to realize a routing mechanism for agents in ad-hoc networks based on pheromones. Agents move in an ad-hoc network spreading pheromone trails during their movements. Such pheromone trails can be used to route messages to agents. Visually, we can think the pheromone trail an agent spread as a long tail. Messages can be routed to the agent by following its tail. More in detail, other than the tuples to create pheromones, we have defined tuples that, once injected, can follow pheromones up to their source. An agent X wiling to send a message to agent Y can wrap the message into one of this follow-pheromone tuples and then inject the tuple.

TUPLES: This application requires two kinds of tuples: *Pheromone* tuples and *FollowPheromone* tuples.

To code a *Pheromone* tuple (Figure 12), one has to inherit from *StrucutreTuple* and realize a maintenance mechanism to let the tuple evaporate after some time. In stark contrast with the field-inspired tuples (*HopTuple* and *FlockingTuple*) these are local tuples. They spread just one-hop away form the source and they have to remain in that place, before evaporating, even after the agent has moved away. Pheromone tuples has been inserted into the TOTA library to be possibly further customized – via inheritance – to create other pheromone-like tuples suited for other specific applications.

To code a *FollowPheromone* tuple, one has to inherit from *StructureTuple* and customize the *decideEnter* method so as the tuple enters only in the nodes having a increasing value of the pheromone (see Figure 13).

```
public class Pheromone extends StructureTuple
{ public int value, VAL, DEC, EVAP;

public Pheromone(int VAL, int DEC, int EVAP) {
 this.VAL = VAL;
 this.value = VAL;
 this.DEC = DEC; // space decay
 this.EVAP = EVAP; // evaporation }

public void makeSubscriptions() {
 SensorTuple st = new
 SensorTuple("<sensor=clock><value=*>");
 tota.subscribe(st, this,"TIME"); }

public boolean decidePropagate() {
  return (value == VALUE); }

public void changeTupleContent() {
 value = value - DEC; }

public void react(String reaction, String event) {
  if(reaction.equalsIgnoreCase("TIME")) {
      value = value -EVAP;
      if(value <= 0) {
         tota.delete(this);
         return;}
  }
}
```

Figure 12: Code of the Pheromone tuple class.

AGENTS: These kinds of agents are rather simple (Figure 14). They wander spreading a pheromone with an ever increasing value, and have a method to send messages to other agents.

```
public class FollowPheromone extends
StructureTuple {
 public int oldVal = 9999;
 Pheromone trail;

 public Pheromone (String msg, String to) {
  content = msg;
  trail = new Pheromone ();
  trail.setContent(to);
 }

public boolean decideEnter() {
  super.decideEnter();
  int val = getPheromoneValue();
  if(val > oldVal) {
   oldVal = val;
   return true;
  }  else
   return false;
 }
}
```

Figure 13: Code of the FollowPheromone tuple class

```
public class PheromoneAgent extends Thread
implements AgentInterface {
 private TotaMiddleware tota;

 public void run() {
  int val = 10;
  while(true) {
   // move randomly
   peer.move(Math.random());
   // while spread pheromone
   Pheromone p = new Pheromone(val, 5, 1);
   p.setContent(peer.toString());
   peer.inject(p);}
  }

 public void send(String msg, String to) {
  FollowPheromone fp = new
   FollowPheromone(msg,to);
  tota.inject(fp);
 }
}
```

Figure 14: Agent example: the agent moves randomly spreading pheromones. Moreover, it has a method to send a message, wrapped in a FollowPheromone tuple, to another agent.

6 CONCLUSIONS AND FUTURE WORK

Stigmergy is getting more and more recognized as a relevant approach for supporting the definition of robust and self-adaptive multiagent systems. Still, little has been done so far to leverage the

practical exploitation and deployment of stigmergy-coordination multiagent systems. The TOTA middleware, by making available a simple API with which to program, in an effective way, a number of diverse stigmergic coordination patterns, proposes itself as a general-purpose approach for programming complex multiagent systems.

Despite the potentials of TOTA, several issues still need to be faced to increase its usability. First, security issues, disregarded in most researches in stigmergic coordination, can no longer be ignored for systems which are to be deployed in open and possibly hostile environments. However, what stigmergy implies in terms of security and privacy is to be fully explored. Second, the lack of an underlying general methodology, enabling engineers to map a specific coordination pattern into the corresponding definition of tuples and of their propagation/maintenance rules, is to be identified. Nevertheless, this is a general drawback of researches on complex MAS, rather than a specific drawback of our approach.

Acknowledgements. Work supported by: the Italian MIUR and CNR in the context of the project "IS-MANET: Infrastructures for Mobile ad-hoc Networks"; and by the Regione Emilia Romagna in the context of the project "LAICA: Laboratory of Ambient Intelligence for a Friendly City".

7 REFERENCES

[1] H. Abelson, et al., "Amorphous Computing", *Communications of the ACM*, 43(5), May 2000.

[2] O. Babaoglu, H. Meling, A. Montresor, "Anthill: a Framework for the Development of Agent-Based Peer-to-Peer Systems", *Proceedings of the 22nd IEEE Conference on Distributed Computing Systems*, Vienna (A), May 2002.

[3] E. Bonabeau, M. Dorigo, G. Theraulaz, *Swarm Intelligence*, Oxford University Press (Oxford, UK), 1999.

[4] C. Borcea, "Spatial Programming Using Smart Messages: Design and Implementation", *24th Int.l Conference on Distributed Computing Systems*, Tokio (J), May 2004.

[5] G. Cabri, L. Leonardi, M. Mamei, F. Zambonelli, "Location-dependent Services for Mobile Users", *IEEE Transactions on Systems, Man, and Cybernetics*, 33(6):667-681, Nov. 2003

[6] 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, 2001.

[7] N. Davies, et al, "L2imbo: A distributed systems platform for mobile computing", *ACM Mobile Networks and Applications*, 3(2):143-156, 2001.

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

[9] P.-P. Grassé, "La Reconstruction du Nid et les Coordinations Inter-Individuelles chez *Bellicositermes Natalensis et Cubitermes sp.* La théorie de la Stigmergie: Essai d'Interprétation du Comportement des Termites Constructeurs", *Insectes Sociaux*, 6:41-84, 1959.

[10] M. Mamei, F. Zambonelli, "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware", *2nd IEEE Conference on Pervasive Computing and Communication*, Orlando (FL), March 2004.

[11] M. Mamei, F. Zambonelli, "Self-Maintained Distributed Tuples for Field-based Coordination in Dynamic Networks", *Concurrency and Computation: Practice and Experience*, 2005, to appear.

[12] M. Mamei, F. Zambonelli, L. Leonardi, "Co-Fields: A Physically Inspired Approach to Distributed Motion Coordination", *IEEE Pervasive Computing*, 4(2):52-61, 2004.

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

[14] J. McLurkin, J. Smith, "Distributed Algorithms for Dispersion in Indoor Environments using a Swarm of Autonomous Mobile Robots", *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, Toulouse (F), 2004.

[15] R. Nagpal, "Programmable Self-Assembly Using Biologically-Inspired Multi-agent Control", *1st Int.l Conference on Autonomous Agents and Multi-agent Systems*, Bologna (I), July 2002.

[16] R. Nagpal, M. Mamei, "Engineering Amorphous Computing Systems", in *Methodologies and Software Engineering for Agent Systems: the Handbook of Agent-Oriented Software Engineering*, Kluwer Academic Publishing (New York, NY), 2004.

[17] V. Parunak, S. Brueckner, J. Sauter, "Digital Pheromones for Coordination of Unmanned Vehicles*", Workshop on Environments for Multi-agent Systems (E4MAS)*, LNAI 3374, Springer Verlag, 2004.

[18] V. Parunak, "Go to the Ant: Engineering Principles from Natural Agent Systems", *Annals of Operations Research*, 75:69-101, 1997.

[19] G. P. Picco, A. L. Murphy, "Using Coordination Middleware for Location-Aware Computing: A Lime Case Study", *Proceedings of the 6th International Conference on Coordination Models and Languages, LNCS No. 2949,* Feb. 2004.

[20] J. Svennebring, S. Koenig, "Building Terrain Covering Ant Robots: a Feasibility Study", *Autonomous Robots*,16(3):313-332, May 2004.

[21] R. Tolksdorf, R. Menezes, "Using Swarm Intelligence in Linda Systems", *Proceedings of the 4th International Workshop on Engineering Societies in the Agents' World*, LNCS No. 3071, 2004.

[22] F. Zambonelli, M.P. Gleizes, M. Mamei, R. Tolksdorf, "Spray Computers: Explorations in Self-organization", *Journal of Pervasive and Mobile Computing*, 1(1):1-20, March 2005.

[23] F. Zambonelli, V. Parunak, "Towards a Paradigm Change in Computer Science and Software Engineering", *The Knowledge Engineering Review*, 18(4), 2004.