Self-Maintained Distributed Tuples for Field-based Coordination in Dynamic Networks

Marco Mamei, Franco Zambonelli
University of Modena and Reggio Emilia
Via Allegri 13, 42100, Reggio Emilia, ITALY
{mamei.marco, franco.zambonelli}@unimore.it

ABSTRACT

Field-based coordination is a very promising approach for a wide range of application scenarios in modern dynamic networks. To implement such an approach, one can rely on distributed tuples injected in a network and propagated to form a distributed data structure to be sensed by application agents. However, to gain the full benefits from such a coordination approach, it is important to enable the distributed tuples to preserve their structures despite the dynamics of the network. In this paper, we show how a variety of self-maintained distributed tuples for field-based coordination can be easily programmed in the TOTA middleware. Several examples clarify the approach, and performance data is presented to verify its effectiveness.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; C.2.4 [**Distributed Systems**]: Distributed Applications.

General Terms: Algorithms, Design

Keywords: Field-based coordination, Distributed data structures, tuple spaces, dynamic networks, mobility.

1. INTRODUCTION

In the near future, computer-based systems will be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of coordinating with each other in the context of complex mobile distributed applications. Such distributed applications will have to face an extreme dynamic scenario. "Agents" (we use this term generically to indicate the active components of a distributed application) can depart out of and arrive into the network at any time, and can roam across different environments. In addition, application's agents will be likely to execute in dynamic, ever-changing networks (e.g., MANETs, sensor networks, etc.).

Coordinating agents' activities in such dynamic networks with traditional models and infrastructures is not easy. It is of great importance to rely on models and middleware infrastructures explicitly designed to take care of all the issues implied in network dynamics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SAC '04, March 14-17, 2004, Nicosia, Cyprus.
Copyright 2004 ACM 1-58113-000-0/00/0000…$5.00.

Among several possible definitions, coordination can also be generally defined as the agents' fundamental capability to decide on their own actions in the context of their operational environment. So, the basic problem of coordination in dynamic network is: *How can this fundamental capability be achieved when the context is dynamically changing?* It is quite natural to see that the cornerstone of this problem is how to dynamically (i.e., at run-time) provide agents with an effective representation of their operational environment (simply *context* from now on). In particular such contextual information must be *(i)* simple and cheap to be obtained; *(ii)* expressive enough to facilitate the definition and the execution of the agents' activities; *(iii)* continuously reflective the actual state of the context.

With this regard, approaches based on the concept of field-based coordination [6] appear very suitable. In these approaches, the agents' context is abstracted and described by means of abstract "fields" distributed in the network and expressing forces driving agents' activities. Agents can locally perceive these fields and decide what to do on the basis of which fields they sense and their magnitudes. This field-based representation of the context is *(i)* simple and cheap to be obtained, since fields are spread in the environment and agents need only to perceive locally the fields in the area, *(ii)* expressive enough to facilitate the coordination activities, since agents need only to combine the perceived fields and be driven by them. Moreover, *(iii)* by letting fields to adapt and self-maintain their distributed shape to reflect changing environmental conditions, field-based coordination can be made robust and adaptive to the above mentioned environmental (e.g. network) dynamism.

From the programmer point of view, the main task in this approach is to map specific coordination problems into suitable field-based representations. To this end, a programming model specifying how to build fields is required. The content of this paper is to present a programming model to program self-maintained distributed tuples, implementing the concept of fields.

The following of this paper is organized as follows: Section 2 describes field-based coordination and specifies a middleware infrastructure suitable to support self-maintained tuples. Section 3 details how to program distributed tuples. Section 4 presents performances and experiments. Section 5 discusses related works and Section 6 concludes.

2. FIELD-BASED COORDINATION

To realize the idea of field-based coordination we basically need: *(i)* to represent fields by means of suitable data structures; *(ii)* an infrastructure holding and supporting these data structures. Given that, field-based coordination is centered on a few key concepts:

1. The agents' operational environment is represented by "field-like data structures", spread by agents and/or by the infrastructure, diffused across the infrastructure, and locally sensed by agents;
2. A coordination policy is realized by letting the agents to locally sense field data structures stored in the infrastructure and act driven by the local shape of these data structures;
3. Both environment dynamics and agents' actions may induce the field data structures to update (aka self-maintain) their values. The new shape is automatically stored in the infrastructure, inducing a feedback cycle (point 2) that can be exploited to globally achieve a global and adaptive coordination pattern.

2.1 Application Scenarios

To clarify the ideas of field-based coordination and to show its applicability, let's focus on some application scenarios.

Field data structures, being spread in a *spatially* distributed environment, are naturally suited in modeling *spatial* information. For instance, a "shepherd" agent could spread in the network infrastructure a field whose value monotonically increases hop-by-hop as it gets farther from it. Such field implicitly enables any other agent, from wherever in the network, of "sensing" the presence of the shepherd and its distance. Also, by sensing the local gradient of the field, each agent could also reach the shepherd by following the gradient downhill. Moreover, if fields are dynamically updated to reflect changes in the system, if the shepherd moves around in the network, its field would be automatically updated and any agent looking for the shepherd would automatically re-adapt its movement. Such an approach is very effective, since it can be trivially coded and adapts to a variety of circumstances (e.g. different networks, nodes' failures) because the fields naturally adapt to the changing environment.

The above is just a specific instance of the more general scenario of field-based motion coordination [4, 7]. In this scenario, the environment is typically described by means of fields attracting agents towards goals and repelling agents from unwanted areas (e.g. dangerous, crowded, already seen areas, etc. depending on the specific application scenario). Fields are modeled by means of distributed data structures having an id and a numeric value, representing the field's magnitude. Moreover, fields are constantly updated to reflect the possibly changing system's situation. Agents simply follow the resulting gradient of these fields, and complex movements are achieved not because of the agents' wills, but because of dynamic re-shaping of the fields.

The behavior of social insects, e.g. ants, has recently inspired some algorithms used to control the activities of multi agent systems [1]. The key of these approaches is in emulating the way in which ants interact one another. They do so by means of pheromone signals they spread in the environment that will be perceived by other ants later on. These pheromone signals can be used to find food sources, or to coordinate efforts in moving some heavy object, etc. Pheromone signals can be easily modeled by means of fields driving agents activities [5]. Instead of being propagated by the infrastructure in all directions, pheromone-field data structures would be distributed by the agents themselves as they move across the network. These data structures would then be used as trials driving agents activities.

Routing in Mobile Ad-Hoc Network (MANET) can be easily modeled as a coordination problem: agents (i.e. network nodes) need to cooperate forwarding each other messages to enable long-range, multi-hop communication. The main principle underlying many routing algorithms is to build a distributed data structure (implemented by means of a set of distributed routing tables) suitable to provide route information. Specifically, this data structure creates paths in the network enabling agents to forward messages in the right direction. Although some routing protocols make the analogy with fields explicit [14], the idea at the basis of distributed routing data structure is the same as field based coordination: provide agents with a ready-to-use representation of the context (i.e. where the message should go next).

Other than these scenarios, field-based coordination has been successfully applied to a variety of other coordination problems: modular robotics [15], amorphous computing [12], videogames [16], showing the general applicability of this idea.

2.2 The TOTA Middleware

The idea of fields can potentially be implemented on any distributed middleware providing basic support for data storing (to store field values), communication mechanisms (to propagate fields) and event-notification mechanisms (to update fields and notify agents about changes in fields' values). However, we think that a set of distributed tuple spaces is a particularly fertile ground on which to build such idea. A set of logically bounded tuples, one in each of the distributed tuple spaces, naturally matches the idea of a field spread across the infrastructure. The content of a tuple would represent the field's physical properties (e.g. magnitude) at a specific location. This is the approach taken by our reference middleware TOTA (Tuples On The Air) [8].

TOTA is composed by a peer-to-peer network of possibly mobile nodes, each running a local version of the TOTA middleware. 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 a specified propagation rule. Specifically, in TOTA, fields have been realized by means of distributed tuples ***T=(C,P)***, characterized by a content ***C*** and a propagation rule ***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 in 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. The spatial structures induced by tuples propagation must be maintained coherent despite network dynamism. 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.

The TOTA API is the main interface to access the middleware. It provides functionalities to let the application to inject and delete tuples in the local middleware (**inject** and **delete** methods), to read tuples both from the local tuple space and from the node's one-hop neighborhood, either via pattern matching or via key-access to a tuple's unique id (**read, readOneHop, keyrd, keyrdOneHop**). Moreover, TOTA supports reactive behaviors by allowing agent to **subscribe** and **unsubscribe** to relevant events (e.g. the income of a new tuple) and to call-back agent's **react** method whenever relevant event happen. Finally, two methods (**store, move**) allow tuples to be actually stored in the local tuple space or to migrate to neighboring nodes. These methods are not part of the main API and are used only within the tuples' code.

3. PROGRAMMING DISTRIBUTED TUPLES

Given the API simplicity, the main task of the programmer is to program distributed tuples. To this end, it is important to have a flexible programming model suitable to specify those distributed data structures. Within the TOTA model, distributed tuples have been designed by means of objects: the object state models the tuple content, while the tuple's propagation has been encoded by means of a specific propagate method.

Following this schema, it has been defined an abstract class *TotaTuple*, that provides a general framework for tuples programming. This is:

```
abstract class TotaTuple {
protected TotaInterface tota;
/* instance variables represent tuple fields */
…
/* 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 */
public void react(String reaction, String event)
 {
}}
```

In TOTA, a tuple does not own a thread, but it is actually executed by the middleware (i.e. TOTA ENGINE) 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 local tuple space. Tuples, however, must remain active even after the middleware has run their code. This is fundamental because their self-maintenance algorithm – see Section 4 – must be executed whenever the right condition appears (e.g. when a new peer connects to the network, the tuples must propagate to this newly arrived peer). To this end, tuples can place subscriptions, to the TOTA EVENT INTERFACE as provided by the standard TOTA API. These subscriptions let the tuples remain 'alive', being able to execute upon triggering conditions.

Although subclassing TotaTuple enables to create every kind of tuple, managing the intricacies of dealing with tuple propagation and maintenance can be tedious and error-prone. To this end, we developed a tuples' class hierarchy (see Figure 1) from which the programmer can inherit to create custom tuples. Classes in the hierarchy take care of dealing with propagation and maintenance with regard to a vast number of circumstances.

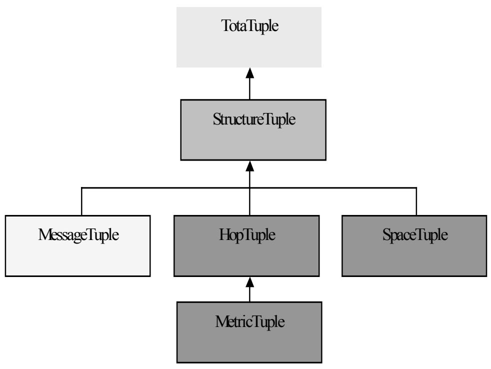


Figure 1: the TOTA tuples' class hierarchy

3.1 StructureTuple

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 still NOT self-maintained. This means that if the topology of the network changes the tuple local values are left untouched.

Such kind of tuples can be used in applications where the network infrastructure is relatively static and thus there is not the need of constantly update and maintain the tuple shape because of network dynamism. Consider a building properly instrumented to have computers in every room and corridor, wired accordingly to the building floor-plan. Each room could inject in the network a distributed tuple providing information on the room itself. The tuple could be propagated in the network, while increasing one integer value, in its content, hop-by-hop as it gets farther form the source. Provided with such tuples, agents in the building could easily get information and reach every room in the building by locally sensing the tuples present in the network.

StructureTuple class inherits from *TotaTuple* and implements the superclass method *propagate* realizing a propagation schema that is at the core of the whole tuples' hierarchy. The code is the following:

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

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. As stated before, 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)*.

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 an example to show the expressiveness of the introduced framework.

A *TimeDecayingGradient* tuple creates a tuple that floods the network in a breadth-first way and have an integer hop-counter that decays with time. To code this tuple one has basically to: *(i)* place the integer hop counter in the object state, *(ii)* overload *changeTupleContent*, to let the tuple change the hop counter at every propagation step, *(iii)* 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, *(iv)* to overload the *makeSubscriptions* method, to let the tuple subscribe to the peer internal clock associating the *TIME* reaction to every clock-tick, *(v)* finally, in the *react* method the reaction is treated by decreasing the integer value and deleting the tuple as soon as the value reaches zero. The code is the following:

```
public class TimeDecayingGradient extends
StructureTuple {
 public String name;
 public int hop = 0;
public boolean decideEnter() {
 super.decideEnter();
 TimeDecayingGradient prev = (
TimeDecayingGradient)tota.keyrd(this);
 return (prev == null ||
        prev.hop > (this.hop + 1));
}
protected void changeTupleContent() {
 super.changeTupleContent();
 hop++;
}
public void makeSubscriptions() {
  SensorTuple st = new SensorTuple("TIME","*");
  tota.subscribe(st,
(ReactiveComponent)this,"TIME");
}
public void react(String reaction, String event) {
  if(reaction.equalsIgnoreCase("TIME")) {
   value = value -1;
   if(value <= 0) {
    tota.delete(this);
    return;
}}}}
```

The rest of the hierarchy of Figure 1 has been built in the same way. Programmers can inherit from the hierarchy by further customizing tuple's propagation to match specific application's requirements.

3.2 MessageTuple

MessageTuples are used to create messages that are not stored in the local tuple spaces, but just flow in the network as sorts of "events". The basic structure is the same as *StructureTuple*, but a default subscription is in charge to erase the tuple after some time passed. Note that it would not be possible to simply remove the *tota.store()* method from the *propagate* method, because previously stored values are used to block the tuple backward propagation. To this end the tuples' value can be deleted only after the tuple wave-front has passed.

It is worth noting that setting the time before deletion is not trivial. If the tuple propagates in a breadth first manner, it can simply be set to the time the tuple wave-front takes to proceed two-hops away. However, if the tuple is propagated in a specific direction and the network topology is closed in a circular track this can lead to a message the continues circulating trough the network endlessly. Up to now, we did not considered this option and the time to delete has been set considering the breadth first case.

Message tuples could be fruitfully applied as a communication mechanism. These tuples, in fact, could embed in their propagation rule the routing policy, without requiring routing agents to properly forward them. Moreover, it is easy to implement with these tuples several different communication patterns like unicast or multicast. Finally, by combining these tuples with *StructureTuples*, it is easy to realize publish-subscribe communication mechanism, in which StructureTuples creates subscriptions paths, to be followed by *MessageTuples* implementing events.

With regard to these tuples, we do not present any examples since we may apply here the same consideration as made in *StructureTuple* case.

3.3 HopTuple

This kind of tuple inherits from *StructureTuple* to create distributed data structures that self-maintain its structure (supported by the TOTA middleware), to reflect changes in the network topology (see Figure 2).

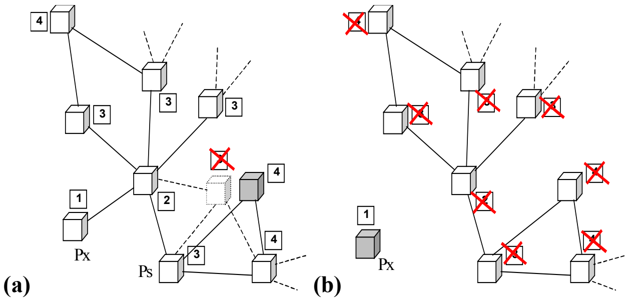


Figure 2: HopTuples self-maintain despite topology changes. (a) the tuple on the gray node must change its value to reflect the new hop-distance from the source Px. (b) if the source detach all the tuples must auto-delete to reflect the new network situation.

Basically this class overloads the empty *makeSubscription* 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 (i.e. an integer *hop* is maintained in the *HopTuple* class).

For example, a *FlockingTuple* creates a data structure that has a minimum at a specific distance (RANGE) from the injecting agent. Such shape self-maintains to remain coherent despite agent movements. This is the code:

```
public class FlockingTuple extends HopTuple {
 private static final int RANGE = 3;
 public int value = RANGE;
 protected void changeTupleContent() {
  super.changeTupleContent();
  if(hop <= RANGE)
   value --;
  else
   value ++;
}}
```

These kinds of tuples are fundamental, since they enable field-based coordination also in presence of dynamic networks. Motion coordination fields like the "shepherd" one presented in 2.1 will be realized with this tuples. The strength of these tuples, from a software engineering point of view, is that agents have simply to inject these tuples and then they can forget about them. All the burden in maintaining their shape is moved away form the agents.

As another example, a *DownhillTuple* creates a tuple that follows the hop counter of a *HopTuple* tuple downhill. To code this tuple one has basically to overload the *decideEnter* method to let the tuple enter only if the value of the *HopTuple* in the node is less that the value on the node from which the tuple comes from. The code to realize this tuple is the following:

```
public class DownhillTuple extends HopTuple {
 public String name;
 public int oldVal = 9999;
 HopTuple trail;

 public DownhillTuple() {
  trail = new HopTuple();
  trail.setContent("ciao");
 }
 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 HopTuple 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++) {
   HopTuple gt =
   (HopTuple)v.elementAt(i);
   if(min > gt.hop)
    min = gt.hop;
  }
 return min;
}}
```

3.4 MetricTuple and SpaceTuple

In some application, it is not possible to rely on a suitable network infrastructure on which to propagate distributed tuples. For example a group of three robots in need to coordinate their movements can create an ad-hoc network, but hop-based distributed tuples would be useless in such a network. In these scenarios it is fundamental to have tuples relying on spatial distances rather than hop-distances. The assumption here is to have location devices installed on nodes or suitable algorithms [12] able to spatially localize neighboring (directly accessible) nodes. In other words, each device must be able to create a local private coordinate system localizing neighborhood nodes.

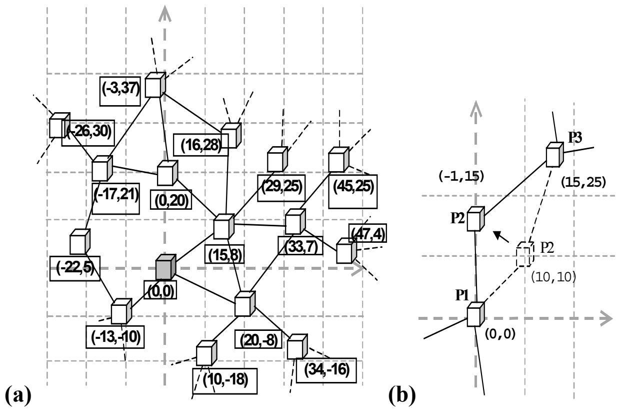


Figure 3: (a) Metric and space tuples create a shared coordinate system, centered in the node that injected the tuple. This is done by having each node combining the coordinate its sees in its private coordinate frame with neighbors' ones. (b) Maintenance of the tuple, requires only local updates, since node that does not move do not change their position with respect to the source (P3 is not affected by P2 movement, since its coordinates with respect of P1 do not change).

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 (see Figure 3(a)). Before explaining the difference between *Metric* and *Space* tuple, it is important to consider that these kinds of tuples are easier to be maintained that *HopTuple*. In fact, once these tuples have been propagated, if a node moves, only its tuple local value is affected, while all the others are left unchanged, since their position with respect to the source does not change. Specifically, a tuple on a node that moves can infer its new value by looking at the tuples stored in neighboring nodes and by considering their coordinates (see Figure 3(b)).

What happens when the source moves? In theory all the tuple instances must be changed because the origin of the coordinate system has shifted. This is exactly what happens in *MetricTuple* where the origin of the coordinated system is anchored to the source node. This of course can lead to scalability problems, especially if the source is highly mobile.

What happens if also the source updates its value looking at neighboring nodes? In this case, the origin of the coordinate system remains where the tuple has been first injected. This, even if no nodes are in that position! The coordinate system is maintained by the network, but not affected by it. This is the *SpaceTuple* implementation.

The following example is a tuple that holds the spatial distance form the source (note that *x,y,z* are maintained in the *MetricTuple* and *SpaceTuple* classes). In the example, *DistanceTuple* inherits form *MetricTuple* and so it always represent the distance from the source. However, if the same tuple would have inherited from *SpaceTuple,* then it would have expressed the distance form the point in which it had been originally injected.

```
public class DistanceTuple extends MetricTuple {
 public int value = 0;
 protected void changeTupleContent() {
  super.changeTupleContent();
  value = (int)Math.sqrt((x*x)+(y*y)+(z*z));
}}
```

4. PERFORMANCES AND EXPERIMENTS

The effectiveness of the field-based approach is of course related to costs and performances in managing fields. Specifically, when considering self-maintained distributed tuples, some fundamental issues arise: what is the cost of propagating a tuple? How much burden self-maintenance adds to the system? Is it scalable? Is the process fast enough?

While the cost of propagating a tuple, for each node, is something inherently scalable, depending only on the number of node's neighbors. The scalability of tuples' maintenance is less clear. To start answering this questions we concentrated on the following: *Are maintenance operations confined within a locality from where a change in the tuple's context originally triggered the maintenance?* If the answer to this question is *yes*, there are good chances that an implementation of this model can lead to good performances: being maintenance confined, it is independent from the size of the network. If the answer is *no*, any implementation is probably doomed to failure.

In the following of this section we are going to present the results we found with regard to the different tuples in the hierarchy of Figure 1. However, before to proceed, a caveat is needed: the following considerations and performances refers only to the classes in hierarchy. It is clear that by subclassing one of these tuples and by introducing complex subscriptions and reactions – that can possibly cause cascading events – performances can change a lot. One of the drawbacks of the flexibility given by our programming model is that the programmer must take care of the performances of the reaction (s)he put into the tuples.

4.1 Structure and Message Tuple

These tuples are not maintained, so the answer to our question is trivially: *yes!* Please note that also with regard of tuples like the *TimeDecayingFloodTuple* described in 3.1, that adds a custom reaction, the answer is still positive, since a change in the tuple context (i.e. time) affects only the tuple itself so it is obviously confined. The same consideration holds also for the delete operation inherent in message tuples.

4.2 HopTuples

With regard to *HopTuple*, answering our question is more complicated. Tuples' 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 triggered whenever, due to nodes' mobility or failures, new links in the network are created of removed. In this context our question becomes: are the tuples' maintenance operations confined to an area neighboring the place in which the network topology had actually changed? This means that, if for example, a device 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 (see Figure 2).

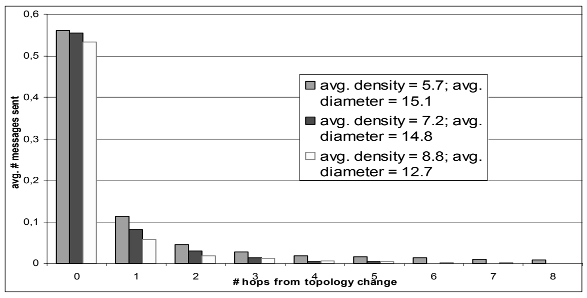
How can we perform such localized maintenance operations in a fully distributed way? To fix ideas, let us consider the case of a tuple incrementing its integer content by one, at every hop, as it is propagated far away from its source.

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.

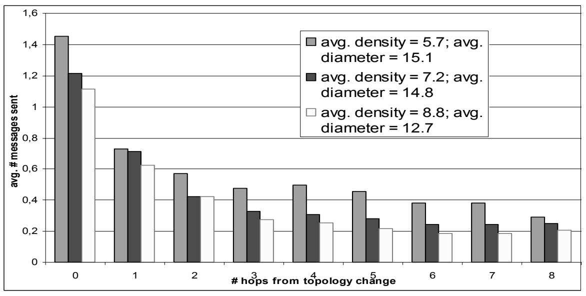
Each local tuple can subscribe to the income or the removal of other tuples belonging to its same type in its one-hop neighborhood.

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 causes 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 2(b)). When a safe-state tuple observe a deletion in its neighborhood it can fill that gap, and reacts by propagating to that node. This is what happens in Figure 2(a), safe-state tuple installed on mode Ps and having value 3 propagates a tuple with value 4 to the hole left by tuple deletion (gray node). 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.



(a)



(b)

Figure 4: 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.

How many information must be sent to maintain the shape of a distributed tuple? What is the impact of a local change in the network topology in real scenarios? To answer these questions we exploited a mobile ad-hoc network emulator running the TOTA middleware, we developed. The graph in Figure 4 shows results obtained by a large number of 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 4(a) shows the average number of messages sent by peers located in an x-hop radius from the origin of the topology change. Figure 4(b) shows the same values, but in these experiment only the source of the tuple moves, changing the topology.

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 4).

This fact supports the feasibility of our 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.

4.3 Metric and Space Tuple

With regard of these tuples answering the question is easy again. In 3.4 we said that Metric tuples maintenance is confined in the node itself for all the nodes apart from the source, it spreads across the whole network if the source moves. So the answer for *Metric* tuples is partially *no!* and for this reason they must be used carefully, maybe with custom rules in their propagation rules limiting a-priori their scope. The answer for *Space* tuple, instead, is a clear *yes!* Since maintenance is strictly locally confined.

All the above considerations are good hints for the feasibility of the model, showing that it can scale to different application scenarios.

5. RELATED WORK

A number of recent proposals rely on different kinds of distributed data structures to support coordination activities in multi agent systems.

Approaches like Lime and XMiddle [11, 13] propose relying on virtually shared data structures as the basis for uncoupled interactions and context-awareness. Each device/agent in the network owns a private data structure (e.g., a private tuple space or a portion of an XML tree). Upon connection with other devices/agents in a network, the privately owned data structures can merge together accordingly to specific policies, to be used as a common interaction space to exchange contextual information and coordinate with each other. By letting the shared data structure mediate and uncouple all interactions, and by not relying on any centralized service, these approaches suit well the dynamics of wireless open networks. However, the acquired information is typically local (deriving from the sharing of data among directly connected devices/agents). Although it is possible to transitively share data across farther devices/agents, it is difficult to acquire a more global perspective and in achieving complex distributed coordination patterns.

The L2imbo model, proposed in [3], 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 agents 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 our idea of distributed tuples are that, in L2imbo, tuples are conceived as "separate" entities and their propagation is mainly performed to let them be accessible from multiple tuple spaces. Our tuples are instead logically bounded and, rather than being used and interpreted a single entities, must be read as part of field-like distributed data structures. Because of this difference, tuples' propagation is defined for every tuple in our approach, while is defined for the whole tuple space in L2imbo.

Smart Messages [2] is an architecture for computation and communication in large networks of embedded systems. Communication is realized by sending in the network messages which include code to be executed at each hop in the network path. The execution of a message at each hop determines the next hop in the path, making messages responsible for their own routing. Smart Messages share with our approach the idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunk of code to determine their propagation. However, in Smart Messages, code is used mainly for routing or mobility purposes. In our approach, instead, tuples are not simply routed though the network, but can be persistent and create a distributed data structure that remains stored in the network. Moreover, in our approach propagation code can also be used to change the message/tuple content, in order to realize distributed data structure and not just replicas.

The research projects Anthill [9] and SwarmLinda [10] share the idea of applying ant-inspired algorithms to Internet-scale Peer-to-Peer systems. Here, field-like data structures – modeling ants' pheromones – instead of being propagated autonomously (e.g. in a breadth-first manner) are spread by the agents as they randomly move across the Peer-to-Peer network. In particular, these data structures create paths connecting peers that share similar files, thus enabling a fast content-based navigation in the network of peers. Clearly, these kinds of systems could be easily modeled within the abstraction promoted by our approach.

6. CONCLUSION AND FUTURE WORK

In this paper we have presented a programming model to create self-maintained distribute tuples over dynamic networks. These kinds of tuples are, in our opinion, a fundamental building block in the context of field-based coordination. What is missing is an underlying general methodology, enabling engineers to map a specific coordination policy into the corresponding definition of tuples and of their shape. Possibly a great number of coordination patterns can be easily engineered even in the absence of a general methodology (e.g., biological systems can be sources of several ready-to-work solutions [1]). Nevertheless, the definition of such a methodology – still lacking in all related approaches based on similar self-organization principles – would be definitely of help and would possibly make our approach applicable to a wider class of distributed coordination problems.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

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

[2] C. Borcea, et al., "Cooperative Computing for Distributed Embedded Systems", 22nd International Conference on Distributed Computing Systems, IEEE CS Press, pp. 227-238, Vienna (A), July 2002.

[3] N. Davies, et al, "L2imbo: A distributed systems platform for mobile computing", ACM Mobile Networks and Applications, 3(2):143-156, Aug., 1998.

[4] S. Johansson, A. Saffiotti, "Using the Electric Field Approach in the RoboCup Domain", RoboCup 2001, LNAI 2377, Springer Verlag, pp. 399-404, Seattle, WA, 2001.

[5] M. Mamei, L. Leonardi, F. Zambonelli, "Co-Fields: A Unifying Approach to Swarm Intelligence", 3rd International Workshop on Engineering Societies in the Agents World. LNAI 2577, Springer Verlag, pp. 68-81, Madrid (Sp), 2003.

[6] M. Mamei, F. Zambonelli, "Field-based Approaches to Adaptive Motion Coordination in Pervasive Computing Scenarios", to be published in "Handbook of Algorithms for Mobile and Wireless Networking and Computing" CRC Handbook, 2004.

[7] M. Mamei, F. Zambonelli, L. Leonardi, "Distributed Motion Coordination with Co-Fields: A Case Study in Urban Traffic Management", 6th IEEE Symposium on Autonomous Decentralized Systems, Pisa(I), IEEE CS Press, pp. 63-70, April 2003.

[8] M. Mamei, Franco Zambonelli, Letizia Leonardi, "Tuples On The Air: a Middleware for Context-Aware Computing in Dynamic Networks", 1st International Workshop on Mobile Computing Middleware at 23rd International Conference on Distributed Computing Systems, IEEE CS Press, pp. 342-347, Providence (RI), May 2003.

[9] O. Babaoglu, H. Meling, A. Montresor, "A Framework for the Development of Agent-Based Peer-to-Peer Systems", 22nd International Conference on Distributed Computing Systems, IEEE CS Press, pp. 15-22 ,Vienna (A), July 2002.

[10] R. Mendez, R. Tolksdorf, "A New Approach to Scalable Linda-systems Based on Swarms", ACM Symposium on Applied Computer 2003, ACM Press, pp. 375-379, Orlando (FL), March 2003.

[11] C. Mascolo, L. Capra, Z. Zachariadis, W. Emmerich, "XMIDDLE: A Data-Sharing Middleware for Mobile Computing", Kluwer, Academic Publishers, Wireless Personal Communications, 21:77-103, 2002

[12] R. Nagpal, "Programmable Self-Assembly Using Biologically-Inspired Multiagent Control", 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna (I), ACM Press, pp. 418-425, July 2002.

[13] G. P. Picco, A. L. Murphy, G. C. Roman, "LIME: a Middleware for Logical and Physical Mobility", 21st International Conference on Distributed Computing Systems, IEEE CS Press, pp. 524-533, July 2001.

[14] R. Poor, "Embedded Networks: Pervasive, Low-Power, Wireless Connectivity", PhD Thesis, MIT, 2001.

[15] W. Shen, B. Salemi, P. Will, "Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots", IEEE Trans. on Robotics and Automation 18(5):1-12, Oct. 2002.

[16] The Sims, www.thesims.com