Programming Pervasive and Mobile Computing Applications: The TOTA Approach

MARCO MAMEI and FRANCO ZAMBONELLI
Università di Modena e Reggio Emilia

15

Pervasive and mobile computing call for suitable middleware and programming models to support the activities of complex software systems in dynamic network environments. In this article we present TOTA ("Tuples On The Air"), a novel middleware and programming approach for supporting adaptive context-aware activities in pervasive and mobile computing scenarios. The key idea in TOTA is to rely on spatially distributed tuples, adaptively propagated across a network on the basis of application-specific rules, for both representing contextual information and supporting uncoupled interactions between application components. TOTA promotes a simple way of programming that facilitates access to distributed information, navigation in complex environments, and the achievement of complex coordination tasks in a fully distributed and adaptive way, mostly freeing programmers and system managers from the need to take care of low-level issues related to network dynamics. This article includes both application examples to clarify concepts and performance figures to show the feasibility of the approach

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; H.4.0 [**Information Systems Applications**]: General; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Pervasive computing, mobile computing, coordination, middleware, tuple spaces, self-adaptation, self-organization

ACM Reference Format:

Mamei, M. and Zambonelli, F. 2009. Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol. 18, 4, Article 15 (July 2009), 56 pages.
DOI = 10.1145/1538942.1538945 http://doi.acm.org/10.1145/1538942.1538945

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

Authors' addresses: Dipartimento di Scienza e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Via G. Amendola 2—42100 Reggio Emilia, Italy; email: {marco.mamei,franco.zambonelli}@unimore.it.

DOI 10.1145/1538942.1538945 http://doi.acm.org/10.1145/1538942.1538945

1. INTRODUCTION

Computing is becoming intrinsically pervasive and mobile [Want and Pering 2005; Abdelzaher et al. 2007]. 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, for example, to support our cooperative activities [Lee et al. 2007], to monitor and control our environments [Borcea et al. 2002; Riva et al. 2007], and to improve our interactions with the physical world [Mamei et al. 2004; Castelli et al. 2007]. 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, that is, strictly related to the environment in which the systems execute, whose characteristics are typically a priori unknown, thus requiring the dynamic enforcement of context awareness; (iii) the adherence to the above requirements must not clash 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 requirements: (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 interoperation; (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 shortcomings, the result is usually an increase in both application and supporting environment complexity.

The approach we propose in this article builds on the lessons of uncoupled coordination models like event-based [Eugster et al. 2003] and tuple-based ones [Cabri et al. 2005], and aims at providing agents with effective contextual information that 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, a tuple can be "injected" into the network from any node and, after cloning itself, can propagate and diffuse across the network according to tuple-specific propagation patterns. Once a tuple is spread over the network, it can be perceived as a single distributed data structure that we call a tuple *field* to draw an analogy with physical fields

(e.g., gravitational), which have different values (i.e., tuples) at different point of space (i.e., network nodes). The middleware takes care of propagating the tuples and of adapting their values (i.e., the global shape of the resulting tuple field) in reaction 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 nearby tuples and associated events (e.g., arrival and dismissing of tuples). This allows agents to perform context-aware coordinated activities.

As we will try to show in this article, TOTA promotes a simple, modular, yet flexible approach to programming distributed applications. The approach is suitable for a variety of scenarios in the area of pervasive and mobile computing, and achieves flexibility without sacrificing the need for a light-weight supporting environment and for acceptable performance and limited overhead. Moreover, the mechanisms proposed by TOTA can support the development of those nature-inspired self-organizing coordination schemes (such as ant foraging, flocking, and self-aggregation) that are increasingly finding useful applications in modern distributed systems [Parunak 1997; Bonabeau et al. 1999; Babaoglu et al. 2006].

The remainder of this article is organized as follows. Section 2 motivates our work, by introducing a case study scenario (which will also act as a working example throughout the article) and by discussing the inadequacy of traditional approaches to pervasive and mobile computing programming. Section 3 introduces the TOTA approach, by overviewing its key underlying concepts and assumptions, and by sketching its current implementation. Section 4 goes into details about the programming of TOTA applications, showing how to define distributed tuples and how to program agents that use the TOTA API and such tuples. Section 5 discusses and analyzes TOTA from a software engineering perspective. Section 6 evaluates TOTA in terms of performance and overhead. Section 7 discussed relates work. Section 8 concludes the article and outlines open issues.

2. MOTIVATIONS AND CASE STUDY

To sketch the main motivations behind TOTA, we introduce a simple case study scenario and try to show the limitations 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. Big museums like the Louvre in Paris host about 8 million tourists per year, with the expectation of reaching 9 million in 2010. This translates on average to about 25,000 tourists per day.[1] These numbers are creating several problems for the current management of services and information in the museum, and a distributed rather than a centralized system architecture could be more

[1]http://www.louvre.fr.

efficient, robust, and less expensive, in particular, a fully distributed approach like the one we will propose in this article.

We assume that each user is provided with a wireless-enabled computer assistant (e.g., a PDA or a smart phone hosting some sort of user-level software agents). 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, other than providing users with wireless connectivity, can be exploited for both the sake of monitoring and control and for providing tourists (i.e., their agents) with information helping them to achieve their goals, for example, properly orienting themselves in the museum, finding specific art pieces, or coordinating their activities and movements with other tourists in a group.

In any case, whatever service has to be provided to tourists in the above scenario, it should meet the requirements identified in the introduction. (i) adaptivity: tourists move in the museum and are likely to come and go at any time. Art pieces can be moved around the museum during special exhibitions or during restructuring works (in big and complex museums like the Louvre about 15% of the rooms on average are closed to the public for different kinds of activities) . Thus, the topology of the overall computational network can change with different dynamics and for different reasons, all of which have to be faced at the application level without (or with very limited) human intervention. (ii) context awareness: as the environment (i.e., the museum map and the location of art pieces) may not be known a priori (tourists can be visiting the museum for the first time), and it is also likely to change over time (due to restructuring and temporary 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: users' PDAs, as well as embedded computer-based devices, may have limited battery life and 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 and mobile computing scenarios. While it is trivial to directly extend the case study from a museum to a campus, or to a stadium or a trade fair, it is also worth noting that warehouse management systems [Weyns et al. 2005], mobile and self-assembling robots [Shen et al. 2004; O'Grady et al. 2005], and sensor networks [Abdelzaher et al. 2007] exhibit very similar characteristics and issues. Therefore, all our considerations can also be applied more generally than just to the case study addressed in this article.

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. Current proposals can be roughly fitted into one of the following three categories: (i) direct communication models, (ii) shared data-space models, and (iii) event based models.

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, for example, via message-passing or in a client-server way. Modern message-oriented middleware systems like, for example, Jini,[2] and most agent-oriented frameworks (e.g., FIPA-compliant ones[3]), support such a direct communication model. One problem with 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, and agents can only perceive and interact with (or request services to) other agents, without any higher contextual abstraction. In the case study scenario, to get information about other tourists or art pieces, tourists have to somehow look them up and directly ask them the required information. Also, to orchestrate their movements, tourists in a group must explicitly keep in touch with each other and agree on their respective movements via direct negotiation. These activities require notable computational and communication efforts and typically end up with ad hoc solutions—brittle, inflexible, and nonadaptable—for contingent coordination problems.

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 [Johanson and Fox 2002], or they can be fully distributed over the nodes of the network, as in MARS [Cabri et al. 2003] and Lime [Picco et al. 2006]. 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 related to nearby art pieces as well as messages left by the other agents. Tourists can easily discover what art pieces are near them and get information about them. However, to locate an art piece further away, 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, each tourist in a group can build an internal representation of the other tourists' positions 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 to negotiate with each other to orchestrate their movements: a lot of explicit communication and computational work is still required by the application agents to effectively achieve their tasks.

[2]http://www.jini.org.
[3]http://www.fipa.org.

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 [Eugster et al. 2003]. Typical infrastructures rooted in this model include REDS [Cugola and Picco 2006] and recent proposals based on structured overlays [Baldoni et al. 2005]. 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 content-based subscription to events). In the case study example, a possible use of this approach could be to have each tourist notify his/her movements across the building to the rest of the group. Notified agents can then obtain an updated picture of the current group distribution in a simpler and less expensive way than required by adopting shared data spaces. However, the fact that tourists can be made aware of the up-to-date positions of other tourists in an easier way does not eradicate the need for them to explicitly coordinate with each other to orchestrate their movements. In fact, even once an agent knows the location of all the other agents, it has still to run some sort of route planning algorithm, taking into account both the building plan and the other agents' location, to meet with them.

The common general problem of the above interaction models is that they are mainly used to gather/communicate only a general-purpose, not expressive, description of the context. Such description is typically based on a number of separate chunks of information that are not easily combined and managed to gather a comprehensive global understanding of the system context. This general-purpose representation tends to be strongly separated from its usage by the agents, forcing them to execute complex algorithms to elaborate, interpret, and decide what to do with that information. On the contrary, if contextual information were represented more expressively and possibly in an application-specific way, agents would find it much easier to adaptively decide what to do. For example, in the case study, if tourists required to meet somewhere in the museum were able to perceive in the environment something like a "red carpet" leading to the meeting room, it would become trivial for them to achieve the task by exploiting such expressive contextual information: just walk on the red carpet.

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 [Cabri et al. 2003] and event-based ones [Cugola and Picco 2006; Eugster et al. 2003] and extends them to provide agents with a unified and flexible mechanism to deal with both context representation and agents' interaction. In general, TOTA enables realizing what could be called "red carpets" that facilitate agents' activities in an environment.

In this section, we present an overview of the TOTA main functionalities and how they can be applied to the case study scenario, and we describe the core components of the middleware. The detailed description of TOTA operations and how to program TOTA applications in detail will be described in Section 4.

3.1 Overview

In TOTA, we propose relying on distributed tuples both for representing application-specific contextual information and for enabling uncoupled interactions among distributed application agents. Unlike traditional shared data-space models, tuples are not associated with a specific node (or with a specific data space) of the network. Instead, tuples are injected and can autonomously propagate and diffuse into the network according to specified patterns. Typically, after injection, a tuple stores itself in the local node, then clones itself and moves to neighbor nodes. This process is repeated recursively. Thus, once a TOTA tuple is spread across the network, it can be regarded as a sort of *field* or a spatial function: the collection of all the tuples created during the propagation of a single injected tuple. Tuple fields are able to serve as messages to be transmitted between application components and also as diffused data structures representing context information.

To support this idea, TOTA assumes the presence of a peer-to-peer network of possibly mobile nodes, each running a local instance of the TOTA middleware. Each TOTA node holds references to a limited set of neighbor nodes and can communicate directly only with them. 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 the nodes' mobility or to their insertions or failures. TOTA tuple fields are spread, stored, and maintained across this network. In other words, the TOTA peer-to-peer network represents the "space" where tuple fields are defined and propagated.

Upon the distributed space being identified by the dynamic network of TOTA nodes, each agent on a node is capable of the following:

—Locally producing tuples and letting them diffuse through the network, in order to publish specific contextual information or to send messages in the network. More specifically, tuples are injected into the system from a node, they are stored in that node, then they are cloned and spread hop-by-hop according to tuple-specific propagation rules (see Figure 1).

—Locally reading tuples in an associative way (i.e., via pattern-matching operations on the basis of a given template) to gather information.

TOTA tuples can be defined at the application level and are characterized by a content **C**, a propagation rule **P**, and a maintenance rule **M**:

$$\mathbf{T = (C, P, M)}.$$

The content **C** is an ordered set of typed elements representing the information carried on by the tuple.

The propagation rule **P** determines how the tuple should be distributed and propagated across the network (i.e., it determines the "shape" of the tuple field). Propagation typically consists in a tuple cloning itself, being stored in the local tuple space, and moving to neighbor nodes recursively. However, different kinds of propagation rules can determine 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

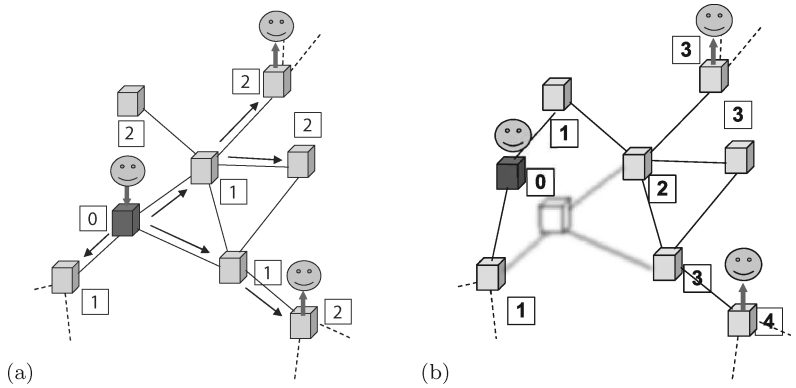


Fig. 1. (a) The general scenario of TOTA: the environment is a peer-to-peer network; application agents live on it and can inject tuples in the network (e.g., the agent on the darker node); tuples propagate in the network according to specific propagation rules, possibly changing their content while propagating. Tuples are represented as squares with their integer content. (b) Upon changes in the network topology (e.g., the darker node has moved), maintenance rules for tuples can update the tuple field to take into account the changed conditions.

the absence of other tuples in the system. In addition, the propagation rule can determine how the tuple's content **C** should change during propagation. Tuples are not necessarily distributed replicas: by assuming different values in different nodes, tuples can be effectively used to build a tuple field expressing some kind of contextual and spatial information. In the end, unlike traditional event-based models, propagation of tuples is not driven by a publish-subscribe schema, but is encoded in tuples' propagation rule. Moreover, tuples can change their content during propagation.

The maintenance rule **M** determines how a tuple field should react to events occurring in the environment or simply to passing time. On the one hand, maintenance rules can preserve the proper spatial structure of tuple fields (as specified by the propagation rule) despite network dynamics. To this end, the TOTA middleware can support tuple propagation actively and adaptively: by constantly monitoring the network local topology and the income of new tuples, the middleware can automatically repropagate or modify the content of 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 tuple field can automatically change to reflect the new topology. On the other hand, tuples (i.e., their content and their distributed structure) can be made variable with time, for instance, to support temporary tuples or tuples that slowly "evaporate."

As an example, the tuple shown in Figure 1 has the following characteristics; (i) the content **C** is a single integer value initialized at zero in the node of injection; (ii) the propagation rule **P** spreads the tuple across the network by increasing the integer value at each network hop; (iii) the maintenance rule **M** takes care of preserving the original propagation structure upon dynamic network reconfigurations.

Clearly, it is possible to exploit tuples for which **P** = **null**, in which case the tuple is only locally stored in the source node (as in a standard tuple space model); tuples for which **M** = **null**, in which case the structure of the tuples remains unchanged after the initial propagation; or tuples in which the maintenance rule specifies deleting the tuple while it is propagating, in which case the tuple simply represents a volatile event.

3.2 The Case Study in TOTA

Let us consider again the museum case study. We recall that we assume that the museum is enriched with a reasonably dense number of fixed wireless devices, for example, those associated with museum rooms and corridors as well as with art pieces, and that tourists are provided with wireless-enabled PDAs to access in a wireless way these embedded devices.

All these devices are expected to locally run the TOTA middleware and, by connecting with each other in an ad hoc network, to form a distributed network of TOTA nodes. In particular, all TOTA devices are connected to close ones only, based on short-range radio connectivity. This also implies a rough form of localization for users: their positions are implicitly reflected by their access points to the TOTA network. However, this does not exclude the presence of more sophisticated localization tools [Hightower and Borriello 2001].

Moreover, we make the additional assumption that the rough topology of the ad hoc network reflects the museum topology. This means that there are no network links between physical barriers (like walls). To achieve this property, we can assume that the devices are able to detect and drop those network links crossing physical barriers (e.g., relying on signal strength attenuation or some other sensor installed on the device). Alternatively, the property can be achieved by assuming that the museum building is preinstalled with a network backbone—reflecting its floor-plan topology—to which other wireless devices connect. This assumption goes in the the direction of having the network map the physical space in the building and, thus, of having the process of propagating a tuple in the TOTA network to result in a configuration coherent with the building plan.

From the actual application point of view, we concentrate on two representative tasks: (i) how tourists can gather and exploit information related to art pieces they want to see; (ii) how they can be supported in planning and coordinating their movements with other, possibly unknown, tourists (e.g., to avoid crowd or queues, or to meet with each other at a suitable location). In this section we will present only a very high-level description of the code. Concrete and detailed code examples will be introduced in Section 4.

3.2.1 *Discovering Art Pieces' Information.* With regard to the first task, TOTA enables a tourist to discover the presence and the location of a specific art piece in a very simple way, and based on two alternative solutions.

—*First solution*. As a first solution, each art piece in the museum can propagate a tuple having as a content the art piece description, its location, and an integer value that, via a propagation rule that increases at each hop, can represent the distance of the tuple from its source (i.e., from the art piece

Art-Piece Tuple

```
C = (description, location, distance)
P = (propagate to all peers
hop by hop, increasing ‘‘distance’’
     element by one at every hop)
M = (update the tuple field, if the museum topology changes)
```

Query Tuple

```
C = (description , distance)
P = (propagate to all peers hop by
hop, increasing the ‘‘distance’’
     element by one at every hop)
M = (delete the tuple after a time-to-live period)
```

Answer Tuple

```
C = (description, location, distance)
P = (propagate following
downhill the ‘‘distance’’ element of the associated
     query tuple. Increment its own ‘‘distance’’ value by one at every hop)
M = (delete the tuple after a time-to-live period)
```

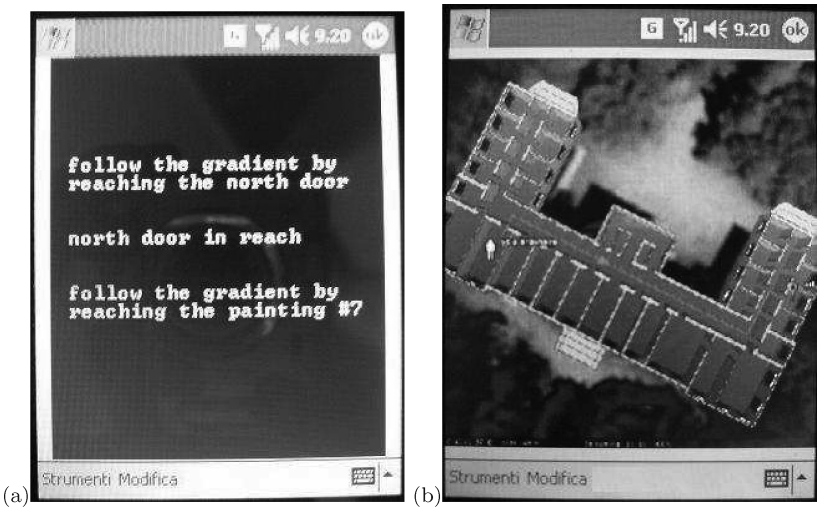
Meeting Tuple

```
C = (tourist_name, distance)
P = (propagate to all peers hop by
hop, increasing the ‘‘distance’’
     element by one at every hop)
M = (update the distance tuple field upon tourist movements)
```

Fig. 2. High-level description of the tuples involved in the museum case study. Art-Piece Tuple is the tuple proactively injected by an art piece to notify other agents about itself. Query Tuple and Answer Tuple are the tuples used by an agent to look for specific art pieces, and used by art-pieces to reply. Meeting Tuple is the tuple injected by the agents into the meeting application.

itself; see Figure 2, Art-Piece Tuple). The tuple fields associated with all the art pieces are thus stored in the TOTA network. Accordingly, each TOTA node (these include users' PDAs) will receive tuples describing all the art pieces in the museum. Any tourist, by simply checking its local TOTA tuple space, can thus discover the presence of such an art piece. Then, by simply following the tuple field backward (i.e., by following the downhill the gradient of the "distance" element), he/she can easily reach the room where the art piece is located without having to rely on any a priori global information about the museum plan.

In this regard, the application agent reads the tuples in neighbor nodes and selects those nodes having a "distance" element lower than its own (i.e., nodes closer to the art piece). Then it suggests the user to move in the direction of the selected node. In particular, a suitable user interface could tell the user the direction to go. If a localization mechanism is available (see Figure 3(b)), the user interface can show the direction to be followed on a map depicted on the user PDA. If no localization mechanism are available (see Figure 3(a)), the agent suggests the user get closer to the node having a lower hop value by naming the node—for example, walk to the "front door" (nodes should have some string description making them identifiable). In either case, following



(a) (b)

Fig. 3. (a) Application GUI without localization—places should be visually identifiable by users. (b) Application GUI with map-based localization.

the agents' advice, the user gets closer and closer to the art piece by following the associated tuple field, until reaching the piece.

—*Second solution*. We could consider that art pieces do not propagate a priori any tuple, but instead sense the arrival of sorts of query tuples propagated by tourists—describing in their content the art pieces the tourists are looking for. Art pieces can then be 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 depicted in Figure 2, Query Tuple and Answer Tuple. Answer Tuple can propagate following the "distance" element of the associated Query Tuple field downhill. In Figure 4(a), the user injects a Query Tuple that spreads across the network. The content of the tuple is a counter that increases at every hop. In Figure 4(b), the art piece that matches the query answers by injecting an "Answer Tuple" that propagates following the gradient of the Query Tuple field downhill. Finally, the application agent on the user PDA could display the retrieved information and direct the user with an approach similar to the one described in the first solution.

These two solutions are presented only for the sake of explaining and clarifying the TOTA approach. In a real implementation setting, the tradeoff between them would be mainly based on the percentage of art pieces being requested by users. If the percentage is low, then the second solution is better, since art pieces that are not requested just stay silent. If the percentage is high, then the first solution is better, since art pieces just propagate their information once.

3.2.2 *Meeting*. Let us now focus on the other task in our case study: how tourists can be supported in planning and coordinating their movements with other, possibly unknown, tourists. Specifically, we focus on a "meeting" service whose aim is to help a group of tourists to dynamically find each other and meet

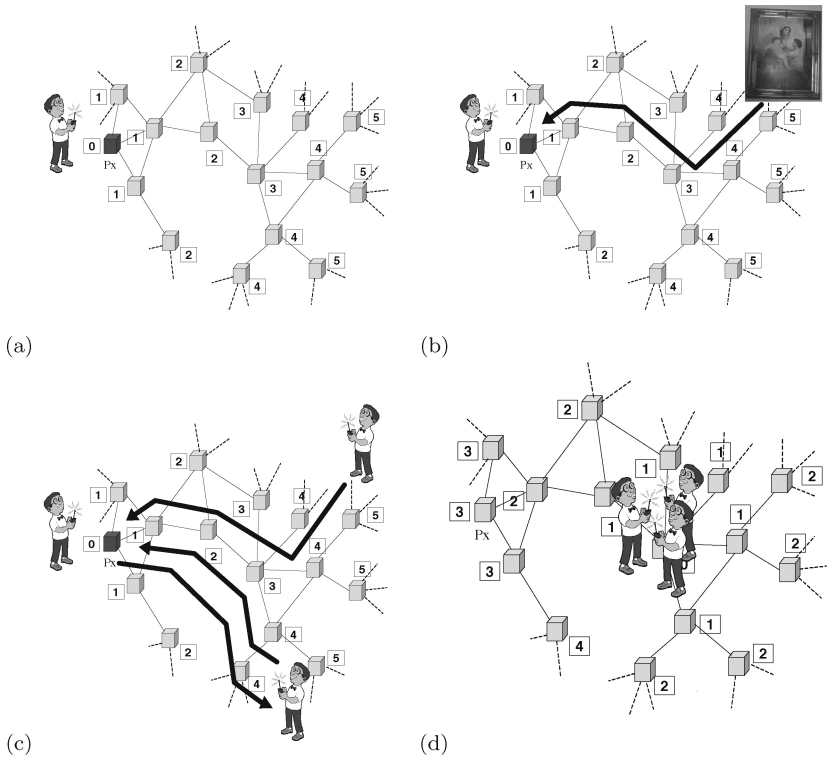


Fig. 4. (a) An agent propagating a Query Tuple to look for a specific information. The tuple propagates creating a gradient routing information back to the source. Numbers represent the tuple with its associated numeric value—defining the gradient (b) A suitable art piece may react to the query by injecting an Answer Tuple that propagates following downhill the Query Tuple field. (c)–(d) Meeting application: for ease of drawing, the figure shows two steps of the meeting process. All agents move toward each other until collapsing in a single room.

in a room of the museum. Different policies can be related to how and where a group of tourists should meet. Here we assume that tourists continuously try to meet the user that is farthest from them. This process recursively leads all the tourists to the same place. This meeting policy can be thought as if all the tourists would attract each other to eventually collapse into a single point. In greater detail, each tourist (i.e., each agent) involved in the meeting can undertake the following actions:

(1) Each agent injects the tuple described in Figure 2 "Meeting Tuple." The tuple propagates across the network to act as a sort of distance field of the agent.
(2) Then, each tourist in the group can start perceiving all such tuples and can start following downhill the gradient of the tuple field with the higher "distance" value. To this end, the application indicates the user where to go with a GUI like the one shown in Figure 3.
(3) This process leads all the tourists to gradually move toward each other until they eventually meet in a room that is suitably located between them.

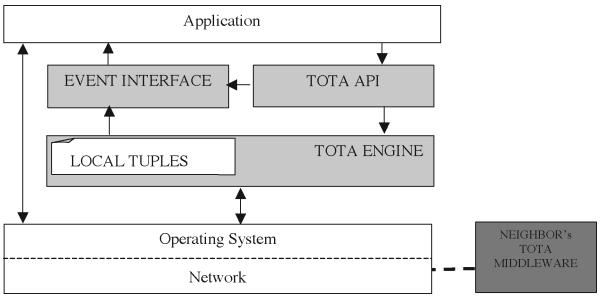


Fig. 5. The TOTA middleware architecture.

In Figure 4(c) and (d), each user follows the gradient of the meeting tuple with highest value (i.e., associated with the farthest user), all of this, again, without having tourists to rely on any a priori agreement or on any knowledge about the museum plan. It is also important to realize that meeting tuples can be programmed to maintain the associated tuple field coherent with the actual positions of their sources (i.e., of tourists). In this way, the resulting tuple fields will be dynamically reshaped as tourists move, which ensures convergence of the process in general and also in the presence of unexpected problems (e.g., a tourist in the group that follows a different route because of crowds in some rooms). This process is also adaptive: since all the users try to reach each other, if one user is delayed (e.g., because of a queue), then the meeting room will be automatically "rescheduled" to one closer to the latest user.

It is worth noticing that TOTA does not provide a mechanism for agent coordination per se. It provides suitable mechanisms and data structures to *support* agent coordination by giving to agents an effective representation of the context (e.g., in terms of the above distance tuple field) and an effective and uncoupled communication mechanism. In particular, the type of context awareness promoted by TOTA is strictly related to spatial awareness. In fact, the tuple fields that spread in the network in the form of TOTA tuples intrinsically enrich the network with some notion of space (e.g., the meeting tuple describes the space by indicating the distance from the user that injected it). In addition, although at the primitive level the TOTA space corresponds to the network space, and spatial distances are measured in terms of hops between nodes, it is also possible to enforce more physically grounded concepts of space. Given the availability of suitable localization mechanisms, the propagation and maintenance rules for TOTA tuples can rely on such localization information. For instance, one can bound the propagation of a tuple to a portion of the physical space simply by having its propagation rule check—as the tuple propagates from node to node—the local spatial coordinates and decide whether to further propagate the tuple or not (see Section 4.2.4).

3.3 The TOTA Middleware

From the architectural viewpoint, the TOTA middleware supporting the above model is constituted by three main parts (see Figure 5): (i) the *TOTA API* is the

main interface between the application agents and the middleware. It provides functionalities to let application agents inject new tuples in the system, retrieve tuples, and place subscriptions to tuple-related and network-related events in the event interface. (ii) the *Event interface* is the component in charge of asynchronously notifying the application agents about subscribed events, like the income of a new tuple or the fact that a new node has been connected/disconnected to the node's neighborhood. (iii) the *TOTA engine* is in charge of maintaining the TOTA network by storing the references to neighboring nodes and of managing tuples' propagation by opening communication sockets to send and receive tuples. This component is in charge of receiving tuples injected from the application level, sending them to neighbor nodes according to their propagation rules, and updating/repropagating them according to their maintenance rules. To this end, this component continuously monitors network reconfiguration, the income of new tuples, and possibly external events.

The TOTA engine also contains a local tuple space in which to store the tuples that reached that node during their propagation. Since tuples propagated in different nodes are not independent but part of a tuple field, the TOTA engine needs a means to uniquely identify tuples in the system. Since the tuple content cannot be used for this purpose (because it can change during the propagation process and because there may be several tuples with the same content), each tuple is marked with an id (invisible at the application level) that is used by the TOTA middleware during tuple propagation and update to trace the tuple. Tuple ids are generated by combining a unique number relative to each node (e.g., the MAC address) together with a progressive counter for all the tuples injected by the node. Moreover, such tuple ids allow a fast (hash-based) accessing schema to the tuples.

From the implementation point of view, we developed a Java prototype of TOTA running on laptops and on PDAs equipped with an 802.11 b wireless interface and a J2ME-CDC (personal profile) Java virtual machine. PDAs connect locally in the MANET mode (i.e., without requiring access points), creating the skeleton of the TOTA network. Tuples propagate through multicast sockets to all the nodes in the one-hop neighborhood. The use of multicast sockets has been chosen to improve the communication speed by avoiding an 802.11 b unicast handshake. By considering the way in which tuples are propagated, TOTA is very well suited for this kind of broadcast communication. We think that this is a very important feature, because it allows implementing TOTA also on really simple devices (e.g., mote sensors [Pister 2000]) that cannot be provided with sophisticated communication mechanisms. As an additional note, and given the intricacies in accessing network events from Java, the current implementation relies on polling at the transport level to get events related to network connections and disconnections and to maintain an up-to-date perspective on the TOTA network.

Since we own only 16 PDAs and some laptops on which to run the system, and because the effective testing of TOTA would require a much larger network, we have also implemented an emulator to analyze TOTA behavior in the presence of hundreds of nodes. The emulator, developed in Java, enables examining TOTA behavior in a MANET scenario, in which the nodes' topology can be

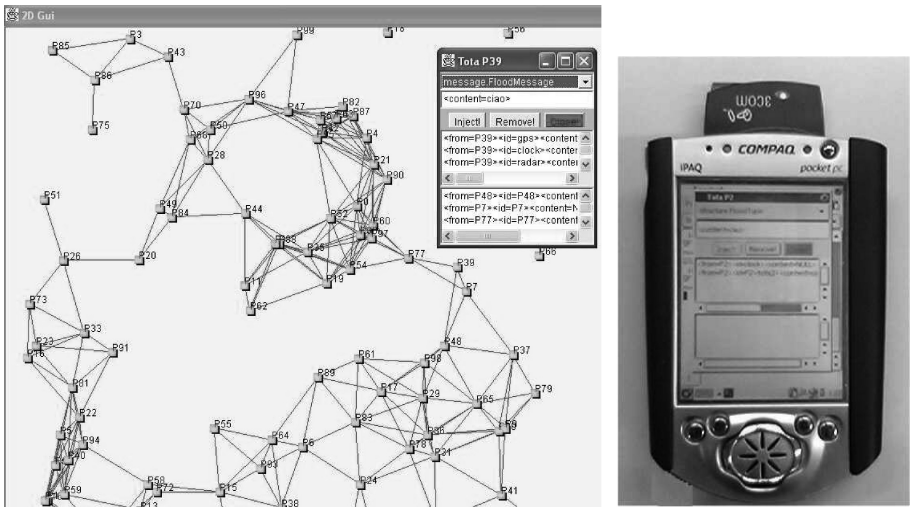


Fig. 6. The TOTA emulator. (left) A snap-shot of the emulator. The snap-shot shows the two-dimensional (2D) representation of the TOTA network. Moreover, a GUI pops up when double clicking on a node, for example, node P39. (right) The same code running on the emulator can be uploaded into the PDA, and it can make available the same GUI.

rearranged dynamically either by a drag and drop user interface or by autonomous nodes' movements. Since we are interested in high-level functionalities, our emulator does not account for MAC-level protocols. Instead, communication facilities are provided on top of an emulated transport level. The strength of our emulator is that, by adopting well-defined interfaces between the emulator and the application layers, the same code "installed" on the emulated devices can be installed on real devices. This allows one to test applications first in the emulator, then to upload them directly into a network of real devices (see Figure 6). Moreover, it enables one also to run the emulator in a kind of "mixed" mode where one or more real PDAs are mapped into specific nodes of the emulator. In this mode, all the PDA communication is diverted into the simulation, providing the illusion that the real PDA is actually embedded in the simulated network and can communicate with its simulated neighbors. In this "mixed" mode, a user with a PDA can interact with both real network peers and emulated ones, and can pretend he or she is immersed in a very large-scale network.

It is worth noting that we also managed to integrate the TOTA emulator with a lot of publicly available third-party simulators to test the effectiveness of our abstractions in many diverse scenarios including robotics [Mamei and Zambonelli 2006a], urban traffic control [Camurri et al. 2006], and non-player-character control in video games [Mamei and Zambonelli 2004]. TOTA code can be downloaded from our Web site.[4]

[4]http://polaris.ing.unimo.it/tota/tota.html.

```
abstract class TotaTuple {
 protected TotaInterface tota;
 /* instance variables represent tuple fields */
 ...
 /* this method initializes the tuple, by giving
 a reference to the current TOTA middleware */
 public void init(TotaInterface tota) {
 this.tota = tota;
 }
 /* this method codes the tuple propagation */
 public void propagate() {}
 /* this method enables the tuple to enforce
 maintenance rules */
 public void react(String reaction, String event){}
}
```

Fig. 7. The main structure of the TotaTuple class.

4. TOTA PROGRAMMING

To develop applications upon TOTA, one has to know the following:

(1) What the primitive operations are that interact with the middleware (i.e., what is the TOTA API).
(2) How to specify tuples, and in particular their propagation and maintenance rules.
(3) How to exploit the above to code context-aware and coordinated activities.

4.1 TOTA API

TOTA is provided with a simple set of primitive operations to interact with the middleware. Most of the operations take as inputs and provide as outputs objects of the class *TotaTuple*. This is the base abstract class for all the TOTA tuples. Concrete implementation of this class will implement the data and methods associated with a specific content (C), propagation rule (P), and maintenance rule (M). Accordingly, in the next subsection we describe first how to create elementary TotaTuple objects. Then we provide a detailed description of the TOTA operations.

4.1.1 *Creating TOTA Tuples.* TOTA tuples have been designed by means of objects: the object state models the tuple content, the tuple's propagation rule is encoded by means of a specific *propagate* method, while a maintenance rule can be implemented by properly coding a specific *react* method. In addition, the *init* method initializes a tuple as it arrives on a node by providing it the reference to the local TOTA middleware, via which a tuple has access to the TOTA API that will be described in the next subsection. This leads to the definition of the abstract class TotaTuple, providing a general framework for programming tuples (see code in Figure 7). The TOTA middleware calls the methods *init* and *propagate* once the tuple enters that node (whether because it has been injected

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

Fig. 8. The TOTA API. TotaTuple arguments are called *templates* in read-like operation because they are used in pattern matching and can contain formal (i.e., null) elements.

by an agent or has moved there because of propagation). The middleware calls the *react* method if the tuple subscribed to a condition that matches a particular event. In particular, we define a *ReactiveComponent* interface exposing the *react* method. Tuples (and also agents) implement that interface.

A detailed description and a number of examples of how to create TOTA tuples (i.e., how to program the *propagate* and *react* methods) is presented in Section 4.2. For now, and for the sake of detailing the TOTA API, we need to specify that all TOTA tuples are constructed by passing an array of objects that will initialize the tuple content:

```
TotaTuple x = new MyTuple(new Object[]{"Hello"});
```

4.1.2 *TOTA Operations.* The operations provided by the TOTA middleware are listed in Figure 8. In this section we will describe each of them by means of sequence diagrams and code examples.

4.1.2.1 *Inject.* The *inject* primitive is used to inject in the TOTA network the tuple passed as the parameter. The operations involved in this method are described in the sequence diagram in Figure 9(top). First the tuple object is passed to the TOTA middleware. Then, the middleware calls the tuple propagation method that will specify whether the tuple will be stored in the local tuple space and if it will be sent to neighbor nodes (see Section 4.2). Once the middleware finishes executing the tuple propagate code on the current node, the inject method returns.

4.1.2.2 *Read and readOneHop.* The *read* primitive accesses the local tuple space and returns a collection of the tuples locally present in the tuple space and matching the template tuple passed as parameter (see the sequence diagram in Figure 9 (middle)).

A template is a TotaTuple in which some of the content elements can be left uninitialized (null). These null elements are the formal parameters of traditional Linda models. In the implemented OO-pattern-matching, a template tuple TMPL and a tuple T match if and only if the following hold:

—TMPL is an instance of either the class of T or one of its superclasses; this extends the Linda model [Gelernter and Carriero 1992] according to object orientation by permitting a match also between two tuples of different types, provided they belong to the same class hierarchy;

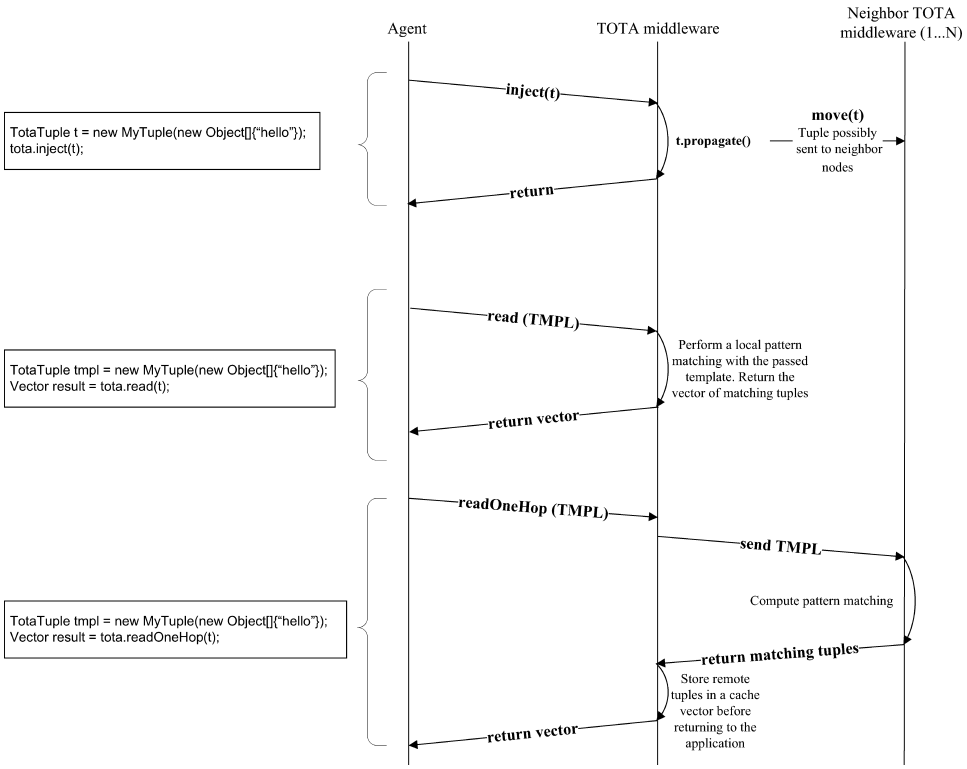


Fig. 9. Sequence diagrams of TOTA operations: (top) Inject method. (middle) Read method. (bottom) ReadOneHop method.

—the not-null elements of TMPL that represent primitive types (integer, character, Boolean, etc.) have the same value of the corresponding elements in T;

—the not-null nonprimitive elements (i.e., objects) of TMPL are equal—in their serialized form—to the corresponding ones of T (two Java objects assume the same serialized form only if each of their instance variables have equal values, recursively including enclosed objects).

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 (see the sequence diagram in Figure 9 (bottom)). The TOTA middleware sends the template tuple TMPL to neighbor nodes. Neighbor nodes compute pattern-matching operations, and return the tuples matching TMPL to the requestor node. All the tuples are collected in a vector that is finally returned to the application agent.

Both read and readOneHop operations are synchronous and nonblocking. They return all the tuples matching the given template, or an empty vector if no matching tuples are found.

4.1.2.3 *Keyrd and KeyrdOneHop.* The *keyrd* and *keyrdOneHop* nonblocking methods are analogous to the former two read methods, but instead of performing a pattern matching on the basis of the tuple content, they look for

tuples with the same middleware-level ID of the tuple passed as argument via a fast hash-based mechanism. These two methods are useful in many situations. For example, to evaluate the local shape of a tuple (i.e., to evaluate the gradient of specific values in the content of the tuple), the agent needs to retrieve the instances of the same tuple in its one-hop neighborhood. keyrdOneHop is a fast method to achieve such a functionality. The sequence diagrams for these methods are equal to the read and readOneHop methods, respectively.

4.1.2.4 *Delete.* The *delete* primitive extracts from the local middleware all the tuples matching the template and returns them to the invoking agent. In this context, it is important to note that the effect of deleting a tuple from the local tuple space may be different depending on both the maintenance rule for the tuple and the invoking agent. In particular, if a tuple field has a maintenance rule specifying to preserve its distributed structure in reaction to events then (i) deleting it from the source node induces a recursive deletion of the whole tuple structure from the network; on the other hand, (ii) deleting it from a different node may have no effect, in that the tuple will be repropagated there if the maintenance rule specifies so. More discussion about maintenance operations and relevant examples may be found in Section 4.2 and in the code in Figure 18 in particular. The sequence diagram for this method is equal to the one of *read* method.

4.1.2.5 *Subscribe and Unsubscribe.* The *subscribe* and *unsubscribe* primitives are defined to manage reactive programming. An agent (or any object implementing the *ReactiveComponent* interface—the second parameter of the method) can subscribe to the insertion and removal of specific kinds of tuples in the local tuple space, and have a suitable *react* method to be called when such events happen (see the sequence diagram in Figure 10). The *subscribeOneHop* method allows one to subscribe to the insertion and removal of specific kinds of tuples in tuple spaces at a one-hop distance.

The class *TsTuple* is used to represent the events happening in the tuple space. In particular, the following tuple represents the event of the tuple x being *inserted* in the TOTA tuple space.

```
TotaTuple x;
TsTuple inPres = new TsTuple(new Object[]{"IN",x});
```

Similarly, the following tuple represents the event of a tuple x being *removed* in the TOTA tuple space.

```
TotaTuple x;
TsTuple outPres = new TsTuple(new Object[]{"OUT",x});
```

TsTuple is basically a wrapper containing the tuple being inserted and removed and an associated label *IN* or *OUT*. IN and OUT are not operations in the Linda sense. They are labels attached to TsTuple to indicate whether the event being represented is an insertion or a removal of a tuple.

TsTuples are temporary objects, automatically created by the TOTA middleware, and used for pattern matching with the stored subscriptions.

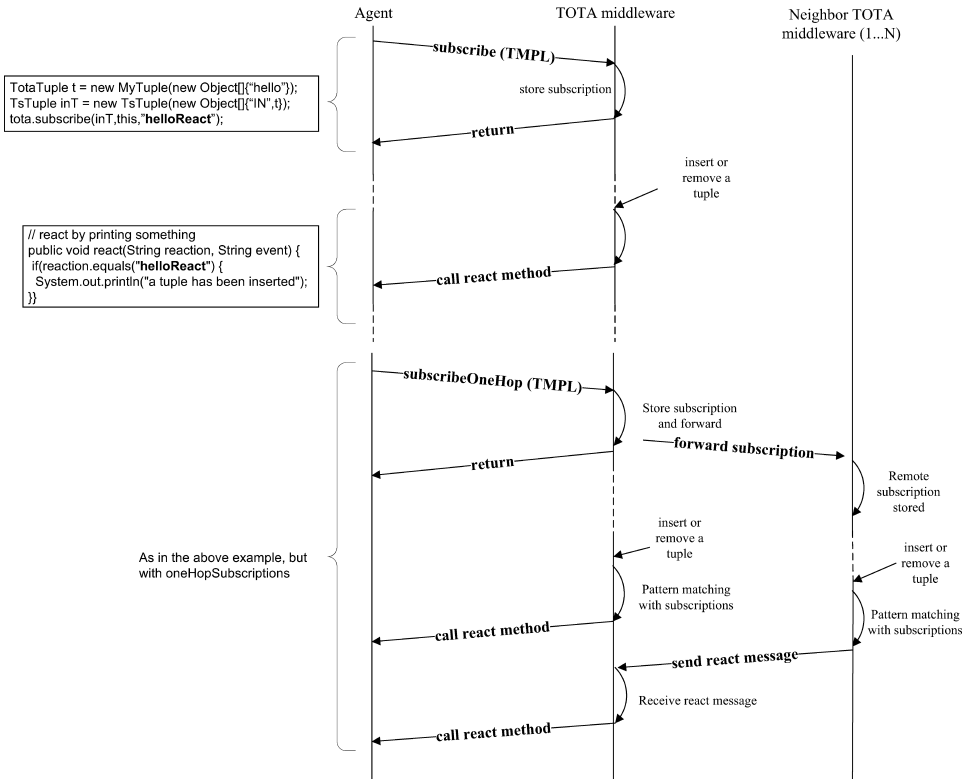


Fig. 10. Sequence diagrams of TOTA operations: (top) subscribe and react methods; (bottom) one-hop subscribe and react methods.

```
// create the tuple to represent an insert event
TotaTuple x = new TotaTuple("<content=*>");
TsTuple inPres = TsTuple(new Object[]{"IN",x});
// subscribe to that event
tota.subscribe(inPres, agent,"IN-X");
...
// react by printing something
public void react(String reaction, String event) {
 if(reaction.equals("IN-X") {
  System.out.println("a tuple has been inserted");
}}
```

Fig. 11. The code to subscribe to events.

Accordingly, nodes can place reactions that will be triggered when the tuple x will be inserted/removed from the tuple space. For example, the code snippet shown in Figure 11 lets the agent print something when a new tuple is inserted in the tuple space. The structure of the code in Figure 11 is rather straightforward. When a tuple matching the tuple x is inserted in the tuple space, the TOTA engine calls the *react* method passing the label *IN-X* used in the subscription, and a string-based serialization of the tuple that triggered the reaction.

Any event occurring in TOTA (including connections and disconnections of peers, and sensor readings) can be represented as a tuple. For example, each TOTA node broadcasts in its one-hop neighborhood a tuple of the class *PresenceTuple* to express its presence. An agent can subscribe to the insertion or removal of these tuples to actually react to peer connections and disconnections. In general, an agent can notify other agents about any kind of event by representing it via a suitable tuple.

The *unsubscribe* primitive removes all the matching subscriptions.

It is worth noting that, despite the fact that all the TOTA read methods are nonblocking, it is very easy to realize blocking operations using the event-based interface: an agent willing to perform a blocking read has simply to subscribe to a specific tuple and wait until the corresponding reaction is triggered to resume its execution.

The operations presented in this section are those that are used by agents using the TOTA middleware. However, it is important to remark that most of these operations trigger a number of other activities performed by the tuples and not by the agents. For example, the inject operation triggers the propagation (and maintenance) procedure performed by a tuple. All these tuples' operations will be described in the following.

4.2 Programming Propagation and Maintenance Rules

One of the most important parts of TOTA is creating tuples and programming their propagation and maintenance rules.

Tuples are created as objects (as described in the previous subsection). TOTA tuples do not own internal threads, but their code—for initialization, propagation, and maintenance—is actually executed by the middleware, that is, by the TOTA engine. However, since tuples must somehow remain active even after the initial propagation, to enable the enforcement of the maintenance rule, the solution adopted by TOTA is to have tuples place subscriptions to the TOTA event interface while propagating (i.e., to place such subscriptions from within the propagate method). Thus, when an event requiring maintenance of a tuple occurs (e.g., when a new peer connects to the network and the tuple must propagate to this newly arrived peer), the react method is triggered to wake up the tuple and maintain it. Maintenance operations are tuple-specific. Each tuple implements its own maintenance in the reactions it defines. In general, a programmer can create new tuples by subclassing the TotaTuple class and by specifying, other than the tuple's content, its propagation rule and maintenance rule with any code.

However, in general, the correct writing of proper propagation and maintenance rules may not be trivial. To facilitate this task, TOTA includes and makes available a whole tuples' class hierarchy (see Figure 12). The classes in the hierarchy already include propagation and maintenance rules suitable for a vast number of circumstances, and make it very easy to create by inheritance custom classes with application-specific propagation and maintenance rules (other than with application-specific content). These classes are integrated in the middleware package; thus it is not necessary to transfer their code upon tuple migration.

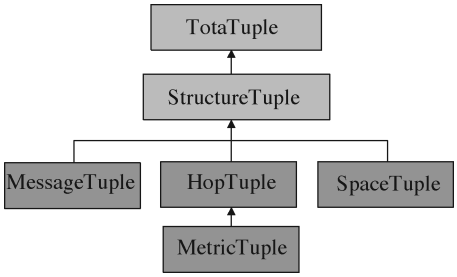


Fig. 12. Some key classes of the TOTA tuples' class hierarchy.

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

Fig. 13. The propagate method in the StructureTuple class.

The key classes in the hierarchy include *StructureTuple*, *MessageTuple*, *HopTuple*, *MetricTuple* and *SpaceTuple*. In the following, we describe in detail these classes showing how they can be effectively exploited to create application-specific tuples. Specifically, for all the classes we will focus on propagation and maintenance operations.

4.2.1 *StructureTuple.* The only child of the TotaTuple class is the class *StructureTuple*. This class is mainly a template to better support the definition of further tuple classes.

The StructureTuple implements the superclass method propagate in order to enforce a diffusion-based propagation method (i.e., a tuple that floods the network). However, it does so with a structured schema that is at the core of the whole tuples' class hierarchy (see code in Figure 13). The so structured propagate method is made *final*: it cannot be overloaded, and all tuple classes have to follow its template to implement specific propagation rules, by overloading the methods by which it is composed. This allow subclasses to redefine such methods without changing the algorithm's structure. We adopted this decision given the modularity and flexibility of such a propagation schema (see also Section 5.4).

The StructureTuple class does not implement any specific maintenance rule. For instance, if the topology of the network changes, the tuple local values are left unmodified. Tuples of this kind can be employed in applications where the network infrastructure is relatively static and thus there is not the need to constantly update and maintain the tuple field distributed structure.

The StructureTuple defines the *propagate* method around the following four methods: *decideEnter*, *decidePropagate*, *changeTupleContent*, *makeSubscriptions*.

In addition, it exploits two methods of TOTA: (i) store is used to actually store the tuple in the local tuple space. (ii) move is used to broadcast the tuple to neighbor nodes. These methods are not used by agents accessing the middleware, but only by tuples in their propagation and maintenance rules.

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 propagate method that, in turn, is composed of the following operations:

—The decideEnter method is the first to be executed. This returns true if the tuple can enter the local node and actually execute there, false otherwise. The standard implementation (diffusion propagation) 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, and false otherwise. The standard implementation of this method returns always true, realizing a tuple that floods the network being recursively propagated to all the peers.

—The method changeTupleContent changes the content of the tuple, and can be used to define how the tuple content should change during propagation. 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 specific events (network events or events occurring in the local tuple space) and enforce maintenance rules that are to be coded in the react method of the tuple. The standard implementation does not subscribe to anything.

—After that, the tuple is stored in the local TOTA tuple space by executing the *tota.store(this)* method. Without this method, the tuple would propagate across the network without leaving anything of itself behind, and no tuple field would ever be formed.

—Then, if the decidePropagate method returns true, the tuple is sent to all the neighbors via the tota.move(this) primitive. With this method, the tuple will be broadcast to neighbor nodes, where the propagation procedure starts again. It is worth noting that the tuple will reach the neighbor nodes with the content changed by the last run of the changeTupleContent method. Because of broadcasting, a tuple cannot define at the network level that should be sent to a specific node. That decision must be made on receipt, based on data held inside the tuple. This implies that the routing decision is moved from the network level to the application layer, affecting the performance. However, application-level routing can result in more flexibility and be compliant with a simple network stack like the one available on sensor network nodes.

Programming a TOTA tuple to create a tuple field basically reduces inheriting from the above class to overload some of the four methods specified above to customize the tuple behavior. A tuple can even query the local tuple space (via the TOTA API) to evaluate the above conditions. We also emphasize that the

```
public class NotMaintainedGradient extends StructureTuple {
 public int hop = 0;
 public boolean decideEnter() {
  NotMaintainedGradient prev =(NotMaintainedGradient)tota.keyrd(this);
  return (prev == null || prev.hop > (this.hop + 1));
 }
 protected void changeTupleContent() {
  super.changeTupleContent();
  hop++;
}}
```

Fig. 14. The NotMaintainedGradient class defines a tuple that floods the network and have an integer hop counter that is increased by 1 at every hop.

presence of both the decideEnter and decidePropagate methods derive from the fact that the decision of whether a tuple should propagate or not can be based sometimes on conditions known at the source node, in which case one can evaluate the condition within the decidePropagate method), but in other cases it can be based on conditions known only at the destination node (like in the above unicast example), in which case one can evaluate the condition within the decideEnter method. This can be used, for example, to code directional propagation: TOTA tuples can also propagate in a given direction instead of radially from the source, for example by having the tuple select in its decideEnter method only those nodes lying in the intended direction. Such a selection can be based on tuples that are already in the recipient node. For example, each node could have a tuple specifying its location. Another tuple could decide to enter only into those nodes within a specific area by reading that tuple. A relevant example of this kind of directional propagation will be presented in Figure 16.

Let us now present an example of how a programmer can define a new tuple class by inheriting from the basic StructureTuple class. The *NotMaintainedGradient* class (see code in Figure 14) defines tuples that flood the whole network and have an integer hop counter that is increased by 1 at every hop. To code this, one has basically to (i) place the integer hop counter in the object state, representing the tuple content; (ii) overload changeTupleContent, to let the tuple change the hop counter at every propagation step; (iii) overload decideEnter so as to allow the tuple to overwrite tuple instances with higher hop counter. This allows one to enforce the diffusion propagation assuring that the hop counter truly reflects the hop distance from the source. We emphasize that one could also think of overloading the decidePropagate method to bound the propagation of the tuple and confine it to a limited number of network hops from the source.

With reference to the case study, it is clear that Art-Piece tuples could be properly represented by instances of the NotMaintainedGradient class, where the integer value defines a gradient that—when followed downhill—leads to the source of the tuple, that is, to the location of the art piece.

However, the NotMaintainedGradient class has no specific maintenance rules: the local instances of the distributed tuple are left unchanged whatever happens in the network and whatever time passes. Clearly, this may create

```
public class MessageTuple extends StructureTuple {
 private int LEASE = 54;
 public void makeSubscriptions() {
  /* get current time */
  SensorTuple st = new SensorTuple(new Object[] {"CLOCK",null});
  st = (SensorTuple)tota.keyrd(st);
  int currentTime = st.value;
  /* create a tuple representing a future event */
  st = new SensorTuple(new Object[] {"CLOCK",new Integer(currentTime+LEASE)});
  tota.subscribe(st,this,"ERASE");
 }
 public void react(String reaction, String event) {
  if(reaction.equals("ERASE")) {
   tota.delete(this);
}}}
```

Fig. 15. The MessageTuple class defines a tuple that floods the network with an integer value and deletes itself after some time has passed, to act as a sort of message that spreads in the network.

problems for applications in dynamic networks (as it can be the case of the case study). This is why the TOTA class hierarchy considers the need for making available specific tuple classes coding suitable maintenance rules.

4.2.2 *MessageTuple.* The *MessageTuple* class is used to create tuples that act as sort of messages that are stored in the local tuple spaces only for a short amount of time before being deleted.

The basic structure is the same as StructureTuple, but a maintenance rule is specified (by properly overloading the makeSubscriptions and the react methods) to erase the tuple after some time passed. The code of the MessageTuple class is shown in Figure 15. It is worth noticing that TOTA takes care of wrapping low-level data (such as the system clock) into a suitable local SensorTuple. This is useful to provide a uniform access to all the resources and information.

We emphasize it is not possible to define such kinds of not persistent tuples by simply removing the tota.store() method from the basic propagation template (as in Figure 13). In fact, it is necessary to temporarily store locally tuples, to prevent backward propagation of tuples. A tuple can be deleted only after the tuple has moved away. Some important remarks are called for in regard to discuss this design decision:

—MessageTuples have to be locally stored to prevent the tuple from moving backward when broadcast. This mechanism to deal with backward propagation may be dangerous (i.e., errorprone). However, problems such as backward propagation (i.e., where a tuple can move and where it cannot) are strictly related to the self-management of tuples (tuples decide where to propagate, delete, or change content on the basis of their own and their neighbor tuples). Some kinds of operations, like avoiding backward propagation, could be managed at the middleware level, but others require application-level information. We decided to keep such issues at the application level to deal with them uniformly and more flexibly.

—Setting the optimal time before deletion is not trivial. If the tuple diffuses in a well-connected network without large holes (i.e., uncovered regions), such time can simply be set to be greater than the time required for the tuple to move two-hops away. However, if the network topology has large "holes" leading to circular paths, a short LEASE time can lead to tuples that endlessly circulate in the network. For this reason, MessageTuples should consider worst-case LEASE time, or should consider confining the propagation distance of a tuple to ensure that propagation will eventually stop.

—The fact that MessageTuples are temporarily stored in the tuple space can be used to subscribe to the incoming of new messages. An agent or another tuple can associate reactions to the insertion of a MessageTuple in the tuple space (see code below):

```
MessageTuple mess = new MessageTuple();
TsTuple inPres =  new TsTuple(new Object[]{"IN",mess});
tota.subscribe(inPres, this,"MESSAGE_ARRIVE");
```

Since the TsTuple is created only once, when the MessageTuple arrives in the node, it induces a reaction operation to be triggered only once, even if the MessageTuples remains in the node for a limited period of time.

Message tuples could be fruitfully applied as a communication mechanism. Subclasses of MessageTuple can be defined to embed specific routing policies in their propagation rule (as in the Smart Messages approach [Riva et al. 2007]). As an example, one can define, by inheritance from MessageTuple, a *DownhillTuple* class that propagates by following downhill the gradient left by other tuples. Specifically, a DownhillTuple tuple follows downhill another tuple whose content is an integer value typically increasing with the distance from the source (e.g., a NotMaintainedGradient tuple). To code this tuple, one has basically to overload the decideEnter method to let the tuple enter only if the value of the tuple being followed (e.g., NotMaintainedGradient) in the node is less that the value on the node from which the tuple comes from (see Figure 16). This is an example of directional propagation, where the tuple propagates toward a specific direction. Clearly the proposed approach of following the gradient can create multiple tuples following the gradient downhill (basically, if a node has more than one neighbor with a lower trail value, the tuple is replicated in such nodes). On the one hand, this increases robustness, since more copies of the tuple will try to reach the destination. On the other hand, this wastes a little bandwidth because of replication. It is worth pointing out that it is possible to devise a mechanism to avoid such replications. Basically it would consist of the downhill tuple selecting one of the downhill nodes before moving to the neighborhood and then specifying in its decideEnter method to enter only the selected node.

By combining DownhillTuples tuples with StructureTuples, it is easy to realize publish-subscribe communication mechanisms, in which StructureTuple creates subscriptions paths, to be followed by MessageTuple implementing events. For instance, this kind of interaction pattern is what we already proposed for the case study: a tourist can propagate queries for art pieces in the form of a "Query Tuple" defined starting from StructureTuple, while art pieces

```
public class DownhillTuple extends MessageTuple {
 public int oldVal = 9999;
 NotMaintainedGradient trail;
 /* note that in the creation of the tuple one must pass the
 reference to the trail tuple to be followed */
 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
 NotMaintainedGradient 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++) {
   NotMaintainedGradient gt = (NotMaintainedGradient)v.elementAt(i);
   if(min > gt.hop)
    min = gt.hop;
  }
 return min;
}}
```

Fig. 16. The DownhillTuple class defines tuples that propagate by following downhill the trial left by another tuple field. Specifically, a DownhillTuple tuple follows downhill another tuple whose content is an integer value typically increasing with the distance from the source (e.g., NotMaintainedGradient).

can react to such queries by injecting an "Answer Tuple" defined starting from DownhillTuple.

4.2.3 *HopTuple.* The *HopTuple* class inherits from StructureTuple the ability to define distributed data structures that are able to self-maintain their propagated structures to reflect dynamic changes in the network topology (see Figure 17, as well as Figure 1), which include changes in the network positioning of the injecting node. To perform this maintenance, HopTuple overloads the empty makeSubscription method of the StructureTuple class to react to any change in the network topology so that—whenever needed—the *react* method can update the hop counter and have it always reflect the current hop distance from the source.

To give readers an idea of how the self-maintenance algorithm works, we provide here an informal description. Let us consider a HopTuple field. Given one of its local tuple instances X on a node, we call another local tuple Y on another node a *supporting tuple* of X if Y belongs to the same tuple field of X, Y is at one-hop distance from X, and the hop-counter value of Y is equal to that of X minus 1. With such a definition, a supporting tuple of X is a tuple

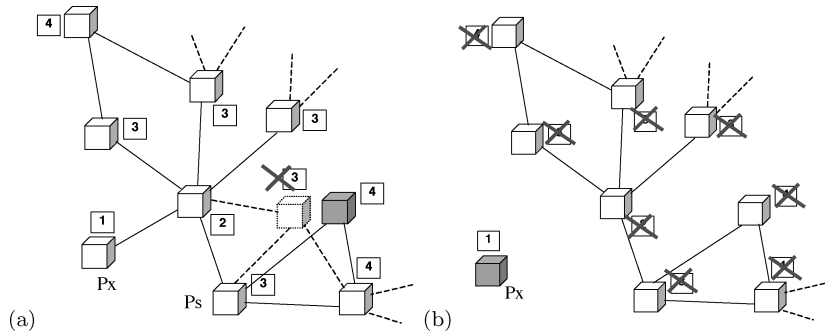


Fig. 17. HopTuples self-maintain despite topology changes. (a) The tuple on the gray node must change its value to reflect the new hop distance from the source *Px*. (b) If the source detaches, all the tuples must autodelete to reflect the new network situation.

that could have created X during its propagation. Moreover, we define X being in a *safe-state* if it has at least a supporting tuple, or if it is in the source node that first injected the tuple (i.e. *hop value* $= 0$). The basic idea of the algorithm is that a tuple not in a safe-state should not be there, since no neighbor tuples could have created it. Therefore, each local tuple must subscribe to the removal of other tuples of its 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 where a tuple is not in a safe-state, it erases itself from the local tuple space. This eventually causes a cascading deletion of tuples until a safe-state tuple can be found, or all the tuples in that connected subnetwork are deleted. On the contrary, in the case where a tuple is in a safe-state, the removal triggers a reaction in which the tuple propagates to that node in order to fix the tuple field.

The code of this algorithm is presented in Figure 18. Readers can refer to Mamei and Zambonelli [2006b] for a more detailed description of the complete algorithm and of its stabilization properties.

The self-maintained tuple fields defined by the HopTuple class are of fundamental importance to support adaptive context-aware activities in dynamic network scenarios, which are the main target scenarios of TOTA. On the one hand, they enable application agents injecting tuples into the networks to fully disregard the actual structure and dynamics of the underlying network. Once the tuple is injected, its maintenance rule ensures that the tuple field structure will always reflect the intended propagation pattern. This moves away from the agent the burden of maintaining the tuple field up-to-date. On the other hand, an agent locally perceiving a tuple is ensured that what it perceives reflects the current situation, and thus is a "live" representation of the context.

In Section 6, we present detailed experiments on the overhead induced by this algorithm. However, we can anticipate that maintenance operations tend to remain confined near the place where the network actually changed because of the multiple paths—especially in rather dense networks—supporting the tuple.

Indeed, all the examples presented above should have better exploited tuples derived from the HopTuple class, in order to tolerate network dynamics and

```
public class HopTuple extends StructureTuple {
 private static final int DEL_DELAY = 5;
 private boolean delayedDelScheduled = false; // the tuple is waiting to delete itself
 public int hop = 0; // fundamental hop counter

 protected void makeSubscriptions() {
  PresenceTuple pres = new PresenceTuple("<peer=*>");
  TsTuple inPres =  new TsTuple("<op=IN><tuple="+pres.serialize()+">");
  tota.subscribe(inPres, this,"PC");
  TsTuple tOut = new TsTuple("<op=OUT><tuple="+this.serialize()+">");
  tota.subscribeOneHop(tOut, this,"OUT");
 }
 protected boolean decideEnter() {
  HopTuple prev = (HopTuple)tota.keyrd(this);
  if(prev!=null && prev.delayedDelScheduled) return false;
  boolean res = (prev == null || prev.hop > (this.hop + 1));
  if(prev!=null && res == true) tota.delete(prev);
  return res;
 }
 protected void changeTupleContent() {
  hop++;
 }
 protected boolean decidePropagate() {
  return !delayedDelScheduled;
 }
 protected void propagate() {
  tota.move(this);
 }
 private void delete() {
  if(!delayedDelScheduled) {
   delayedDelScheduled = true;
   tota.unsubscribe(new Tuple("<content=*>"), this);
   // local message acting as a zombie indicator. Zombie tuples cannot support other tuples
   LocalMessage m = new LocalMessage();
   m.setContent("<content="+this.id+">");
   tota.inject(m);
   int currentTime = getTime();
   SensorTuple st = new SensorTuple("<sensor=clock><value="+(currentTime+DEL_DELAY)+">");
   tota.subscribe(st, this,"DELAY_DELETE");
  }
 }
 public void react(String reaction, String event) {
  super.react(reaction,event);
  if(reaction.equalsIgnoreCase("PC") && !delayedDelScheduled)
   // safeState method is not reported. It works as reported in the text.
   if(safeState() && decidePropagate()) tota.move(this);
  else if(reaction.equalsIgnoreCase("OUT") && !delayedDelScheduled) {
   if(!safeState()) delete();
   else if(decidePropagate()) tota.move(this);
  }
  else if(reaction.equalsIgnoreCase("DELAY_DELETE")) {
   delayedDelScheduled = false;
   tota.unsubscribe(Tuple.deserialize(event),this);
   tota.delete(m);
   tota.delete(this);
  }
}}}
```

Fig. 18. A HopTuple maintenance code.

```
public class BoundedTuple extends HopTuple {
 private int MAX-HOP = 3;
 protected boolean decideEnter() {
  return hop <= MAX-HOP && super.decideEnter();
}}
```

Fig. 19. A BoundedTuple creates a tuple that propagates within MAX-HOP hops from the source. The code of this tuple is trivial since it exploits the superclass methods.

dynamic movement of the source. Let us consider the meeting tuple in the case study (see Figure 2 "Meeting Tuple"). This can be directly realized via the HopTuple class, so that, whenever a user moves in the museum, the propagated meeting tuple (which expresses its presence and its distance) will always be perceived by other agents as reflecting the current position of the user (with some slight delay due to the time required in the propagation of the tuples and dynamic updates). What is important to note here is that, if all users decide to exploit meeting tuples to meet with each other by following downhill the sensed meeting tuples, the resulting meeting process would be adaptive, and would resemble a natural process of self-aggregation [Babaoglu et al. 2006].

Additionally, it is worth noting that once the HopTuple code is available, the coding of other self-maintained tuples becomes simple. For example, one can think of bounding the propagation of tuples to a limited number of hops, by simply overriding the decidePropagate method. This does not affect the algorithm for self-maintenance, which continues performing its work in the subpart of the network in which the tuple is intended to be propagated. This kind of tuple is presented in Figure 19. Thus, even if the code in Figure 18 is rather complicated, it can be effectively leveraged by other programmers to code their own application-specific tuples.

4.2.4 *MetricTuple and SpaceTuple.* All the tuple classes presented above rely on propagation patterns directly related to the underlying network structure, that is, based on network hops. However, in several application scenarios, possibly even in our case study, it may be helpful to ground tuple propagation on more physically grounded concepts of space (e.g., meters) rather than on network distances.

The *MetricTuple* and *SpaceTuple* classes of the TOTA class hierarchy tackle this problem to support the propagation and maintenance of TOTA tuples based on the actual localization of TOTA nodes (and accordingly require nodes provided with a localization device). In particular, both these tuple classes define three float numbers *(x, y, z)* as a content, representing the physical coordinates in a three-dimensional reference space. Once one of these tuples is injected into the network, the *(x, y, z)* tuple is set to *(0, 0, 0)*, and the tuple propagates by having its content change so that the *(x, y, z)* tuple always reflects the coordinates of the local node in a coordinate system centered where the tuple was first injected (see Figure 20). In other words, these tuples define a coordinate system centered around the source node.

Tuple maintenance is rather simple. Once a node moves (and given this is not the source node), only its tuple local value is affected, while all the others

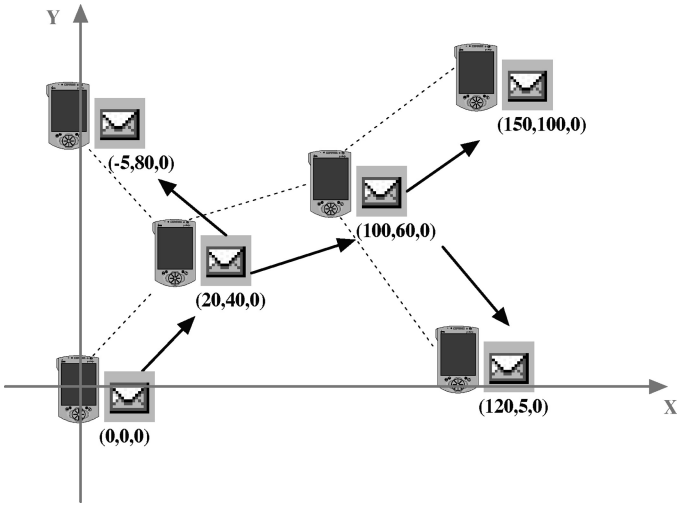


Fig. 20. MetricTuple and SpaceTuple tuples create a shared coordinate system, centered in the node that injected the tuple.

are left unchanged. In fact, the other nodes' physical positions with respect to the source do not change.

The key difference between MetricTuple and SpaceTuple tuples relates to what happens when the source moves. For MetricTuple tuples, the origin of the coordinate system of the injected tuple is anchored to the source node, and when the source moves all the coordinates in the propagated tuple field will be updated. This of course can lead to performance problems: if the source is highly mobile, this will imply a continuous update of the whole tuple field structure. For this reason, the maintenance rule of MetricTuple tuples comes with a threshold value that defines a minimal distance of movement of the source before an update of the tuple field structure is triggered. SpaceTuple tuples solve this problem in a more radical way: for these tuples, the origin of the tuple coordinate system is anchored with the initial position of injection, so that if the source moves the distributed tuple structures remain the same, and only the source node will have to update its local coordinate values.

To present an example, a *DistanceTuple* is a tuple that holds as a variable in its content the spatial distance from the source (which can be easily computed from the *(x, y, z)* tuple.) DistanceTuple, since it inherits from MetricTuple, always represents the distance from the source, even when the source moves. However, if the same tuple would have inherited from *SpaceTuple*, then it would have expressed the distance from the initial injection point (see code in Figure 21).

Considering again the museum application scenario, a DistanceTuple could be used for the meeting service to provide a better spatial resolution than the HopTuple. Moreover, if DistanceTuple is inherited from SpaceTuple, it would be possible for a user to indicate a meeting point and then move away. The meeting point would be maintained also if there were no nodes at that location.

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

Fig. 21. A DistanceTuple is a tuple that holds the spatial distance from the source.

Clearly, starting from the above tuple classes, it is trivial to derive classes that enforce the same propagation and maintenance rules but rely on a different coordinate system (e.g., a polar or geographic coordinate system).

4.3 Programming Agents

The last step involved in programming TOTA applications is coding the agents that use the TOTA API and create the instances of TOTA tuples to achieve the needed application goals.

For the sake of simplicity, and without loss of generality, we detail how to program the agents involved in the museum case study to gather contextual information and to enforce the meeting process.

As sketched out in Section 3.2, in this case study, an agent running on the user PDA provides information by looking at the gradient of some tuples distributed in the environment, and by using such gradients for the sake of both routing the propagation of other tuples and directing user movements.

On the one hand, the propagation of tuples following the gradient of other tuples has been presented in Figure 16. On the other hand, as introduced in Section 3.2, an agent can direct user movements by indicating the direction where to go.

In the following, we present the part of the agent code that takes most advantage of TOTA API.

4.3.1 *Gathering Contextual Information.* The first solution proposed in Section 3.2 considers the following: (i) each art-piece agent (i.e., the agent residing on an art piece) propagates in the network a tuple describing itself; (ii) the tuple will propagate across the whole network actually broadcasting the information about the art piece; (iii) tourist agents can simply query the local tuple space to discover the presence of an art piece and its estimated location. This is a kind of push-based approach in that the information about art pieces (i.e., tuples) is pushed to the tourist PDA.

The code for these two agents is shown in Figures 22 and in 23, respectively. What we can see is that it is extremely simple for the programmer to exploit the TOTA class hierarchy to define tuples with some specific content, for example, as those necessary to describe a piece of art in the MyGradient class, and then to use the TOTA API to inject tuples (as in the art agent) or to use the tuple class as a template to locally retrieve tuples (as in the tourist agent).

The same example can be easily translated to work in different scenarios, for example, for outdoors. A tourist visiting a town could exploit the same approach to discover the presence of monuments, churches, or whatever, by locally querying for tuples distributed in the town. To this end, the only modification

```
public class ArtAgent1 implements AgentInterface {
 private TotaMiddleware tota;
 public void run() {
  /* create and inject the description */
  MyGradient description = new MyGradient(new Object[] {"Monna Lisa", "Da Vinci", "1492", "Room X"});
  tota.inject(description);
}}

class MyGradient extends HopTuple {
 // define the tuple content
 public String title, author, location;
 public int year;
 public MyGradient (Object[] o) {super(o);}
}
```

Fig. 22. ArtAgent first solution. This agent will run in every art piece in the museum.

```
public class TouristAgent1 implements AgentInterface {
 private TotaMiddleware tota;
 public void run() {
  /* create a template to query the local tuple space */
  /* the template match all art pieces by Da Vinci */
  MyGradient template = new MyGradient(new Object[] {null, "Da Vinci", null, null});
  /* query the local tuple space
  Vector v = tota.read(template);
  /* at this point, the user agent can analyze all tuples he has
  got in return and decide what to visit, also based on
  information about distances of the art pieces.
  In the code below, the agent displays the title of all found
  art pieces closer than 2 hops */
  for(int i=0; i<v.size(); i++) {
   MyGradient mg = (MyGradient)v.elementAt(i);
   if (mg.hop<=2) System.out.println (mg.title);
}}}
```

Fig. 23. TouristAgent first solution. This agent runs on the user PDA.

suggested for the code of art-piece agents is that of exploiting, instead of the NotMaintainedGradient or HopTuple classes, the SpaceTuple class. This enables tourist agents to analyze physical distances, which is definitely more suited to outdoor scenarios. In addition, for such outdoor scenarios, it can make sense to bound the propagation of art-piece tuples to, say, a few kilometers.

The second approach sketched in Section 3.2 considers that art-piece agents do not propagate tuples, but are instead programmed to sense the incoming of query tuples propagated by tourists and to react by propagating backward to the requesting tourists their location information. Although such a solution induces slower response times to tourists, it has the advantage of limiting the spreading of tuples in the network, and may thus be suggested for the retrieving of noncritical contextual information.

Such a second approach is coded by the *ArtAgent2* and the *TouristAgent2* classes presented in Figures 24 and 25, respectively.

Each art-piece agent is identified by a description representing the art piece it stands for. It subscribes to the tuples querying for the art pieces they represent, that is, to the income of any tuple MyGradient matching its own description. The reaction to such an event is to inject a MyDownhillTuple, which simply inherits from DownhillTuple to specify a content with the same

```
public class ArtAgent2 implements AgentInterface {
 private TotaMiddleware tota;
 private Object description[];
 private MyGradient tempquery;
 /* agent body */
 public void run() {
  /* subscribe to the query */
  description =  new Object[] {"Monna Lisa", "Da Vinci", "1492", "Room X"}
  tempquery = new MyGradient(description);
  tota.subscribe(tempquery,this,"answerQuery");
 }
 /*code of the reaction, here it injects the
 answer tuple. The answer will be coded by a
 MyDownhillTuple following the query.*/
 public void react(String reaction, String event) {
  Object params =  Object[] {tempquery, description};
  MyDownhillTuple answer = new MyDownhillTuple(params);
  tota.inject(answer);
}}
```

Fig. 24. ArtAgent second solution (ArtAgent 2). This agent will run every art piece in the museum.

```
public class TouristAgent2 implements AgentInterface {
 private TotaMiddleware tota;
 private Object descriptiontemplate [];
 /* agent body */
 public void run() {
  /* inject the query */
  descriptiontemplate = new Object [] {"Monna Lisa", null, null, null};
  MyGradient query = new NMGradient(descriptiontemplate);
  tota.inject(query);
  /* subscribe to the answer: the answer will be
  conveyed in a MyDownhillTuple */
  MyDownhillTuple answer = new MyDownhillTuple(descriptiontemplate);
  tota.subscribe(answer,this,"DISPLAY");
 }
 /* code of the reaction that simply prints out the result */
 public void react(String reaction, String event) {
  if(reaction.equals("DISPLAY")) {
   gui.show(answer);
}}}
```

Fig. 25. TouristAgent second solution (TouristAgent 2). This agent runs on the user PDA.

structure of that of MyGradient. Such a MyDownhillTuple propagates by simply following backward the query tuple to reach the tourist agent which issued the request.

A tourist agent, by its side, performs just two simple operations: it injects into the network a tuple of class MyGradient, filling it with a partial content describing the art piece the user is looking for. Then the agent subscribes to the income of all the MyDownhillTuple tuples (which are assumed to describe an art piece and its location) having the partial content formerly specified in the query (“Mona Lisa”). The associated reaction displayReaction is executed on receipt of such a tuple to print out the full content of the received event tuple in the user interface.

```
class MeetingGradient extends HopTuple {
 String username;
 public MeetingGradient (Object[] o) {super(o);}
}

public class MeetingAgent extends Thread implements AgentInterface {
 private TotaMiddleware tota;
 public void run() {
  /* inject the own meeting tuple to
  participate the meeting */
  MeetingGradient mt = new MeetingGradient(new Object [] {"MyName"});
  tota.inject(mt);
  while(true) {
   /* read all other agents' meeting tuples via a null template */
   MeetingGradient coordinates = new MeetingGradient(new Object [] {null});
   Vector v = tota.read(coordinates);
   /* evaluate the gradients via a proper getDestination method,
   and select the peer to which the gradient goes downhill */
   Point destination = getDestination(v);
   /* suggest the user to move downhill following
   meeting tuple via an appropriate user interface*/
   gui.show(destination);
}}}
```

Fig. 26. Agent example: MeetingAgent. This agent runs on the PDA of all the users in the meeting group.

Also for this second solution, we can consider the use of MetricTuple tuples for outdoors.

In addition, for both the proposed solutions, it may make sense to adopt temporary tuples, that is, tuples with a limited time to live (as already specified in Section 4.2.2 and according to the maintenance rules of Figure 15), so as to avoid leftovers of no longer meaningful tuples.

4.3.2 *Meeting and Motion Coordination.* With regard to the meeting application, the algorithm followed by *MeetingAgents* (see code in Figure 26) is very simple: agents have to determine the farthest peer, and then move (better: suggest to their user where to go) by following downhill that peer's presence tuple. In this way agents will move toward each other, meeting in a room between them. To this end, each agent injects a MeetingGradient tuple, inherited from HopTuple and enriched with a description of the user, to notify other agents about its location. Then it will read the MeetingGradients injected by the other agents, extract the one corresponding to the farther agent and display the direction to go to follow the tuple downhill. The result is an orchestrated movement of tourists, all of which will eventually be guided to a proper location for the meeting, without requiring agents to know anything a priori about the topology of the museum and the locations of other users.

Starting from this simple example, and always exploiting similar kind of tuples, a variety of other patterns of coordination can be enforced to orchestrate the activity of tourists. For example, a tourist guide could inject a tuple similar to the MeetingGradient to be easily reachable by tourists (which could follow the tuple downhill to reach the guide). As another example, the guides themselves could try to follow other tuples uphill in order to stay as far as possible

from each other, and thus improve the coverage of the museum. As an additional example, TOTA could be used to easily implement self-organized load balancing of crowds within the museum. To this end, we can assume that, for each room, a *Crowd* tuple is spread across the museum with a tuple field whose value represents the number of people in that room. Following this Crowd tuple downhill, tourists could try to stay away from crowded areas and queues while visiting the museum. Further details on these and other motion coordination examples can be found in Mamei et al. [2004].

In conclusion, all such patterns can be easily extended to other application scenarios (e.g., outdoors) and to robots or computer-enriched objects.

5. SOFTWARE ENGINEERING ANALYSIS

The goal of this section is to highlight some peculiar aspects of TOTA making it effective from a software engineering point of view.

5.1 Context Representation

The main benefit of TOTA from a software engineering point of view is that TOTA tuples provide a form of context representation that is directly usable by the agents. This is because the distribution of TOTA tuples implicitly defines specific usage patterns with which the information will be accessed.

For example, a MeetingGradient tuple encodes both the context information allowing a user to be reachable by other people in the meeting group, and it implicitly defines the follow-the-gradient usage pattern to use such context information (see Figure 26). Such tuples will be application specific in that they encode context information that is relevant and applicable to a given task.

This is a rather different approach from those surveyed in Section 2.2. In direct-communication, shared data-space, and event-based models, the idea is to make available to agents all the context information in a rather general-purpose and "raw" format (e.g., the (x, y) position of other agents in the meeting group). Once all the relevant information has been collected, the agents need to process it and decide what to do.

Context information represented with such standard techniques is more general purpose and reusable than information represented via TOTA tuples. For example, (x, y) positions of agents can be used also in many applications other than in the meeting one. However, the drawback of this standard approach is that it is much more difficult for agents to act on the basis of such information. With the context representation supported by TOTA, instead, the agent actions become trivial and driven—literally—by the context tuples.

5.2 Separation of Concerns

Another benefit of TOTA is promoting a strong separation of concerns between the tasks of representing and communicating data and context information among agents, and the tasks related to the application business logic that agents have to undertake. The first task is achieved by means of TOTA tuples; the second is coded in the agent program.

(1) To code TOTA tuples, the programmer focuses on specific context information (including information about other agents and about the surrounding environment), and on specific usage patterns with which the information will be accessed. In principle, the ideal situation would be to have a vast library of tuples encoding a wide range of context information with several usage patterns to be used across several application contexts.
(2) To code agents, the programmer focuses on the business logic of the application (i.e., what the agent has to accomplish). The programmer selects the TOTA tuples that best represent the context information the agent has to expose, and codes how agents *use* the received tuples. For simple applications, like the meeting one, the use of TOTA tuples is trivial, since the tuples already encode suitable usage patterns. For complex applications, the programmer has to decide how to combine different tuples and how to use them profitably. For example, some tuples may be of use only in some stages of the application, or in combination with other tuples.

TOTA tuples enable application-specific coordination, in that tuples can be created with different content and rules for different application needs. However, they do not enforce application-level coordination: TOTA tuples are basically an efficient mechanism to encode context information in a convenient and effective way, that is, to encapsulate and separate from the business logic some low-level logic that would have been otherwise spread in the agent code. The separation of concerns enforced by TOTA basically moves complexity out of the agent and into the tuple code. However, in the process, the overall complexity of the application is reduced because entities have clearly defined responsibilities and goals.

5.3 TOTA Tuples and Self-Organization Mechanisms

By adopting a natural metaphor, tuple fields resulting from the propagation of tuples can be considered as sorts of virtual force fields or as sorts of chemical pheromones. For example, a tuple field with a numeric value decreasing with distance from the source can mimic the concentration of chemicals in biological cells, or it can mimic the strength of physical fields in space [Nagpal 2002]. As another example, tuple fields can be the basis of pheromone-like data structures driving agent activities [Bonabeau et al. 1999]. Pheromones can be realized via tuples locally deposited by agents as they move across the network. A maintenance rule decreasing the numeric content of the tuple can emulate pheromone evaporation.

This characteristic enables one to easily reproduce in TOTA those self-organization phenomena that are increasingly finding useful applications in modern distributed system scenarios. There the key coordination activities revolve around the idea of acting on the basis of various types of distributed data structures (e.g., pheromone trails and virtual force fields) [Parunak 1997; Bonabeau et al. 1999; Babaoglu et al. 2006]. Since such data structures can be easily coded via TOTA tuple fields, TOTA can be considered as a middleware

providing the proper abstractions to support the general-purpose development of a wide range of self-organization algorithms that are currently developed from scratch with ad hoc technical solutions.

Indeed, some of the tasks in the case study, as resolved in TOTA, are already very similar to some examples of the above bio-inspired mechanisms. The meeting task, involving agents attracted toward each other, is similar to those chemotaxis mechanisms allowing bacteria to move in a coordinated way [Nagpal 2002; Parunak 1997]. The routing mechanism involved in art-piece discovery is instead similar to ant-based routing [Bonabeau et al. 1999], where ants follow pheromone trails to reach food.

5.4 Tuple Programming

Tuple programming is one of the central parts of TOTA. To improve programming effectiveness and convenience, the core classes have been built on the basis of two design patterns [Gamma et al. 1999]. In particular, TOTA mainly relies on the *command* and the *template method* patterns.

The command pattern is used so that the agent injecting a tuple is completely decoupled from the other nodes in the network; in fact, all the instructions on how to propagate and handle the tuple are coded into the tuple itself.

The template method pattern has been used to code the StructureTuple class that is the basis of the TOTA tuples' class hierarchy (see Figure 13). In fact, the propagate method in the StructureTuple class defines the skeleton of the propagate algorithm deferring some steps to subclasses. Consequently, it lets subclasses redefine such steps without changing the algorithm's structure.

The above two patterns simplify the coding of tuple propagation, since it should always be clear which method to subclass to achieve a given tuple structure. Moreover, TOTA tuples use the same middleware API of agents. This notably eases application development in that developers use the same mechanisms both for agents and tuples.

From an algorithmic viewpoint, one of the most important parts of tuple programming is the coding of tuple maintenance. Most of the application-specific maintenance rules are fairly simple to implement. They involve tuples to delete or change their values upon specific local conditions. For example, a *Number-Of-People-To-Meet* tuple could change its value according to the number of distinct MeetingGradient tuples that are in the local tuple space. The tuple should subscribe to the incoming and departure of MeetingGradient tuples in the local tuple space, and should react by increasing or decreasing its value. The maintenance rule of tuples involving cascading reactions (e.g., the HopTuple) is instead more complicated. Still, by subclassing from the already provided classes, most of the issues are already taken care of. For example, it is worth noting that maintenance problems dealt with in the HopTuple code are really those problems arising in most of the tuples whose values relate to the topology of the network. Thus the code can be leveraged in all such kinds of network-based situations.

Table I. Time Costs in a PDA

Propagation Time on a WiFi PDA (IPAQ 400 MHz)	
T_{prop}	99.7 ms
T_{send}	67.2 ms
T_{travel}	0 ms
T_{rcv}	21.2 ms
T_u	188.1 ms

6. EXPERIMENTAL EVALUATION

To complement the above software engineering analysis, it is important to analyze the effectiveness of TOTA from a distributed system point of view. In particular, TOTA effectiveness is related to the costs, performance, and overhead involved in propagating and maintaining TOTA distributed tuples.

We want to emphasize that in these analyses we are interested in measurements at the middleware and application levels. For example, we will count the number of messages being exchanged by nodes, or the number of operations being executed by nodes. Finer-level analysis, dealing with MAC-level protocols, packet size, etc., are out of the scope of this article and would limit our discussion to a specific network technology.

6.1 Tuple Propagation and Deletion

Tuple propagation is the key operation in TOTA, together with tuple deletion (which, after all, is simply a different form of propagation).

In general, the multihop algorithm for propagating or deleting a tuple is something inherently scalable from the individual nodes' viewpoint. In fact, each node has to propagate a tuple (or propagate deletion requests) only to its immediate neighbors. Thus, whatever the size of the network, the effort on the part of TOTA nodes is constant. The issue of scalability for maintaining the coherence of tuples' distributed structures and for sustaining the propagation of tuples in intensive applications will be discussed later on.

The time costs involved in propagating or deleting a tuple derive from several contributions. Let us focus on the basic operation of a generic StructureTuple traveling from a node to another neighbor node. The total cost T_u involved in such a process accounts for several contributions: $T_u = T_{rcv} + T_{prop} + T_{send} + T_{travel}$. T_{rcv} is the time to receive, deserialize, and have the tuple ready to execute the propagate method. T_{prop} is the time taken by a tuple to run its propagate method on a node. T_{send} is the time required to serialize and send the tuple content. T_{travel} is the time needed to let the stream of data arrive on the other node.

These values as measured in our PDA implementation of TOTA are reported in Table I.

In an ideal case, if the above were the only costs to be accounted for, a tuple would propagate at a distance of X hops in a $X \cdot T_u$ time. However, in practice, the propagation time tends to be greater. In fact, tuple propagation does not happen always in a perfect expanding-ring manner (a general problem of flooding in ad hoc networks; see, e.g., the studies reported in Ganesan et al. [2002]).

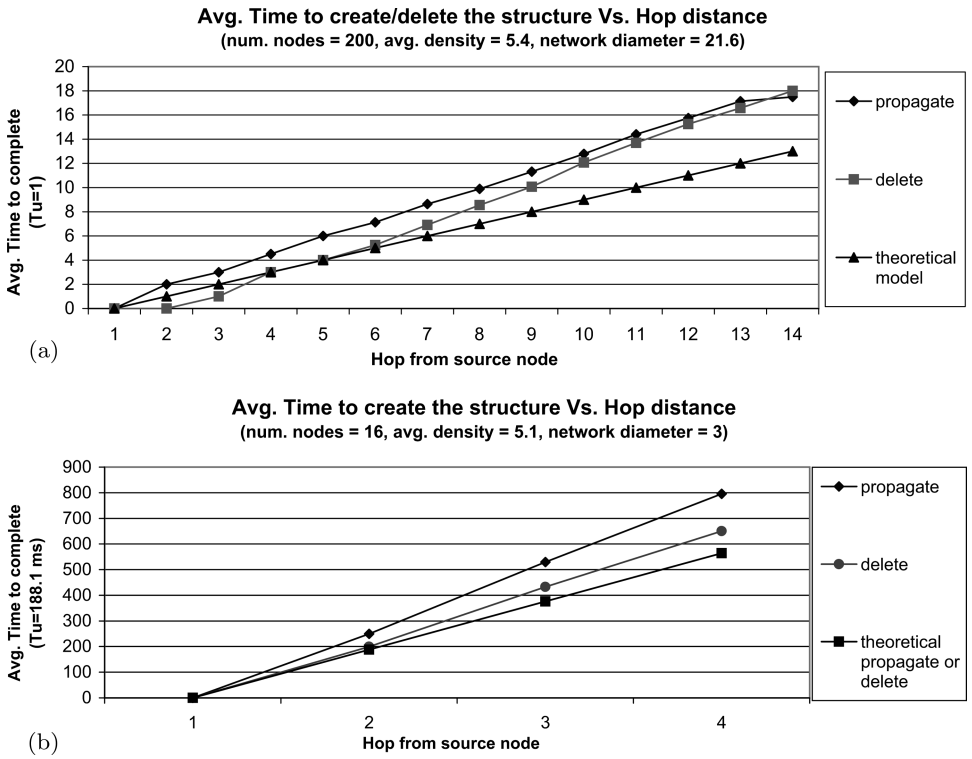


Fig. 27. (a) Time required to propagate and delete a TOTA tuple in simulation experiments. (b) Time required to propagate a TOTA tuple in a MANET of real PDAs.

The correction of imbalances and of backward propagations in tuple propagation requires some extra operations (and thus extra time) that are difficult to extract and analyze.

To assess these costs, we performed several experiments both with our TOTA emulator and with a network of real PDAs. On the one hand, we first simulated a network with 200 nodes connected in a MANET. A randomly selected node injects several TOTA tuples in the network. The time taken by the tuples to reach a distance of x-hops away from the source has been recorded and averaged together over 100 experiments. The results are depicted in Figure 27(a) (where we assume a reference $T_u = 1$). On the other hand, we set up a real MANET consisting of 16 PDAs connected with each other so as to obtain a network with a diameter of three hops. Then we propagated tuples in there measuring delays. The real time to propagate and delete a tuple in the PDA network, averaged over a number of experiments, is reported in Figure 27(b). In both the experiments, it is easy to see that the number of operations required was slightly greater than the ideal model, though still inherently linear with the number of hops. The same results held in the case of the tuple deletion process, but deletion was slightly more efficient than propagation. This is because deletion can remove tuples without taking into consideration the constraints in the tuple propagation methods (e.g., decideEnter, decidePropagate).

The standard deviation associated with these experiments was rather low (i.e., about 10–20% of the mean). This is rather normal in that only local propagation or deletion glitches can let the tuple deviate from the theoretical model.

To get the real meaning of these results, one can consider that, for a network of nodes with a wireless radius of 20m (which could be the typical radius of WiFi in a cluttered environment), the physical speed for tuple propagation would be around 75 m/s, that is, 270 km/h. Such a speed enables one to effectively adopt TOTA in scenarios involving humans and robots, all of which exhibit speeds notably slower than that of tuples, and thus making the delay at which they would perceive information (in the form of tuples being propagated) mostly negligible.

6.2 Self-Maintenance Algorithms

Analyzing and evaluating the performances of the self-maintenance algorithms is a bit more complex, but it is nevertheless very important to assess the feasibility of our approach and its scalability.

In this subsection, we evaluate self-maintenance algorithms for the different classes of the TOTA hierarchy shown in Figure 12. These classes are already integrated in the TOTA middleware. In particular, the discussion on HopTuple will require more space because its self-maintenance algorithm is more complicated than the others.

In any case, we emphasize that our results apply only to the algorithms already implemented in the classes of the TOTA hierarchy, which we expect to be reusable in the vast majority of cases. Subclassing from them to implement different self-maintenance algorithms is always possible, but may introduce totally different behaviors and performance. It is clear, in fact, that adding maintenance reactions to the tuples (especially in the case of subscriptions involving remote tuples) may notably affect the performance, due to the costs in letting nodes communicate information about new tuples being inserted and removed.

6.2.1 *Overhead and Scalability.* Let us now focus on the problem of verifying how the performance and the overhead induced by tuple maintenance algorithms affect the nodes of the network, and whether such performance and overhead are independent of the network size and, thus, are scalable.

StructureTuple tuples are not maintained: once propagated, they do not add any further burden to the system.

MessageTuple tuples have a maintenance rule consisting of a reaction that deletes the tuple after some time has passed. This reaction affects all the nodes in which the tuple propagates, possibly even far nodes, but only once. So it does not introduce critical performance issues.

The issue of self-maintenance is instead critical for HopTuple tuples. Consider a tuple of this class propagated to the whole system, and assume that a change in the network topology occurs somewhere requiring the execution of some maintenance operations. Then, should such operations affect the whole network (even far nodes), that would make it not feasible to generally adopt

such tuples in the presence of dynamic networks. On the other hand, if maintenance operations are confined within a locality from where the triggering event occurred, then events at distant nodes do not accumulate with each other, and distant nodes are not affected by events occurring somewhere else. In this case, the impact of the self-maintenance algorithm would be localized and, thus, scalable.

Establishing by analysis how the HopTuple self-maintenance algorithm impacts on the network is very complex, in part because the actions performed by it strongly depend on the network topology [Mamei and Zambonelli 2006b] (see Figure 17). Thus, to assess the impact, we exploited the TOTA emulator and performed a large number of experiments to measure the scope of maintenance operations.

To perform the experiments, we ran several simulations varying the node density and their initial position. In particular, we ran six sets of experiments where we randomly deployed 200, 250, 300, 350, 400, and 450 nodes in the same area, thus obtaining an increasing node average density and a slightly shrinking network diameter. All the experiments were repeated 100 times with different initial network topologies to average the results. Specifically, we conducted two classes of experiments.

6.2.1.1. *Experiment* 1. In this experiment, we wanted to verify the number of operations required to fix a HopTuple tuple when the movements of random nodes change the network topology. We measured the number of operations in terms of the number of application-level messages exchanged by nodes to maintain the tuple field structure.

In the experiments reported in Figure 28(a), a randomly chosen node injected a HopTuple in the network. After that, randomly chosen nodes started moving independently, perturbing the network. In particular, a randomly picked node moved randomly for a distance equals to one wireless radius in a random direction (in the experiment, only one node moved at a time). This movement changed the network topology by creating and disrupting links. The number of messages sent between nodes to adjust the TOTA tuple, according to the new topology, was recorded. Specifically, we evaluated the average number of messages exchanged by nodes located x-hops away from the moving node. For example, if there were three nodes at one-hop distance from the moving one, exchanging one, zero, and zero messages, respectively then, the first value of the graph would be 0.33.

The experiments reported in Figure 28(b) were conducted in the same manner. This time, however, nodes moved for a distance of one-quarter of a wireless radius to show what happens for small-topology reconfigurations.

In general it is worth pointing out that the length of movements is not very significant in that a long movement can be always divided into a sequence of short movements. Moreover, the motion pattern being taken also does not influence the reconfiguration effort, which is only determined by how the topology of the network is perturbed. In theory, it would be sufficient to measure the operations required to fix the tuple distributed shape every time a network link is created or disrupted. For these reasons, these experiments could be generalized

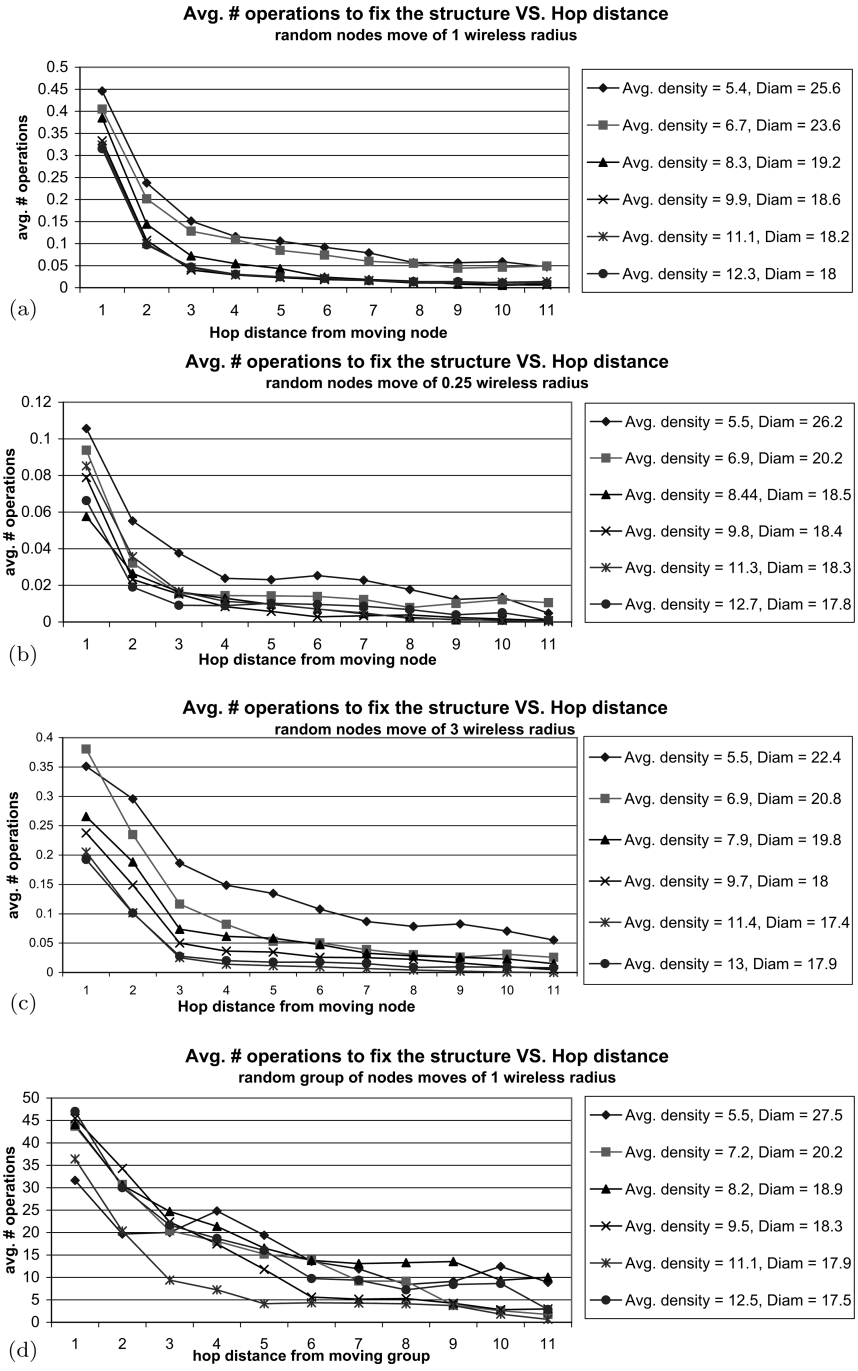


Fig. 28. The number of maintenance operation for HopTuple tuples decreases sharply with the hop distance when topology reconfigurations are caused by (a) random node movements for one wireless radius, (b) random node movements for a quarter wireless radius, (c) random node movements for three wireless radii, and (d) a group of nodes moving for one wireless radius.

to other kind of movements with different motion patterns. To, test this idea, we repeated the experiment with nodes moving three wireless radii obtaining similar results (see Figure 28(c)).

In the last set of experiments, nodes were not free to move independently from one another. They were grouped with some of their neighbors and, once a node moved, all the nodes belonging to the group moved also. More specifically, in the reported experiment, a randomly selected node moved for a distance of one wireless radius and all the nodes directly connected to it (i.e., in its one-hop neighborhood) moved together. This case is particularly significant in that it can represent the situation in which a group of devices, carried by a person or in a car, moves together because physically bounded. The results of this experiment are shown in Figure 28(d). We repeated this experiment also with larger groups of nodes (i.e., when a node is going to move, all the nodes within two or even three hops move together), obtaining analogous results.

Looking at all these figures, the most important consideration we can make is that, when a node moves and the network topology consequently changes, a lot of update operations are required near the area where the topology changes, while only a few operations are required far away from it. This implies that, even if the network and the tuple field being propagated have no well-defined boundaries, the operations to keep the field shape consistent are confined within a local scope. This fact supports the idea that the operations to fix distant concurrent topology changes do not add up, making the approach scalable in term of network size and concurrent movements.

Despite these positive results, it is fair to report that the standard deviation of the experiments results tended to be quite large (i.e., about 70–80% of the average). This was due to the inherent random nature of network reconfigurations: sometimes a node at the edge of the network moves without perturbing a lot the topology of the network, while sometimes an highly connected node at the center of the network moves and completely transforms the topology of the network. Thus, although on average the cost of maintaining a tuple is locally confined, for some critical topology changes, maintenance operations also spread to distant nodes.

An important remark with regard to these topics relates to the time interval within which a node probes its neighborhood to detect connections and disconnections of other nodes. All the topology changes happening in between the probe interval will be perceived as an atomic reconfiguration by the node. Accordingly, the number of reconfiguration efforts and thus the performance may change notably. In all the experiments, the probe interval was rather high so that reconfigurations were perceived as atomic events. For this reason, experiments 28(a)–28(c) had almost the same costs because in the network being considered a movement of one or three hops do not change the entity of the reconfiguration dramatically. On the contrary, if in the three-hop movement (Figure 28(c)) nodes would have probed their neighbors three times more frequently, we would expect to have triple the maintenance operations (result of Figure 28(a) times 3).

Accordingly, the probe time should be set to the highest value compatible with the requirements in regard to the responsiveness of the application.

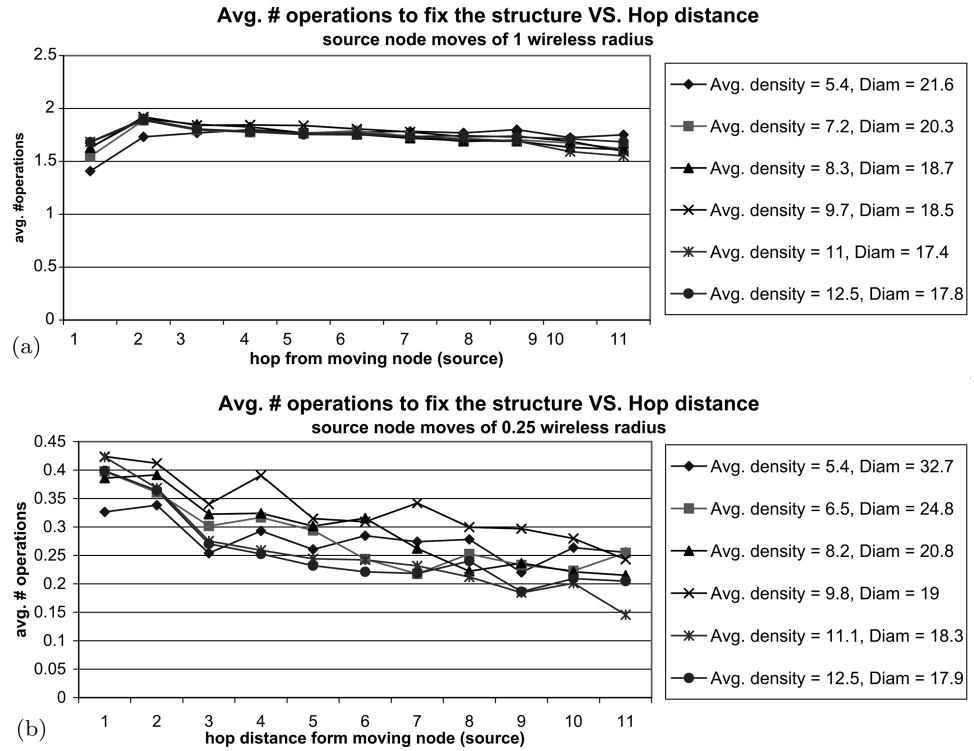


Fig. 29. The number of maintenance operations for HopTuple tuples slightly decreases with the distance from the source even in the worst-case scenario when the source moves. (a) The source node moves for one wireless radius. (b) The source node moves for a quarter wireless radius.

6.2.1.2. *Experiment* 2. This array of experiments was similar to the previous one, but this time only the source node that injected the HopTuple tuple moved, altering the network topology. This is a worst-case scenario, because when the node that injected the tuple (source node) moved, major (and not localized) changes were expected in the tuple field. In Figure 29(a) the source node moved randomly of a distance of one wireless radius. In Figure 29(b) it moved of one-quarter of a wireless radius.

Looking at the figures (especially the one related to small movements), it is again possible to see that the number of operations slightly decreased with the distance from the topology change, but such a number remains relevant even at high distances. This implies that, in scenarios where the nodes that inject HopTuple tuples are highly mobile, self-maintenance can introduce a rather big overhead on the system. For this reason, for highly mobile source nodes, it may be worthwhile limiting the propagation distance of tuples.

The standard deviation associated with these experiments was much lower than that the previous case (i.e., about 30–40% on average). This is because the movement of the source node tended to create a large perturbation in the tuple field independently of where the source was located in the network.

It is fair to point out that this is exactly the case of the museum application, where the agents that inject the meeting tuple are the ones that move toward each other. To lower maintenance costs, we set up a rather large interval to probe the neighborhood to detect network changes. On the one hand, following this approach, a lot of reconfiguration effort is saved. On the other hand, the meeting task is not undermined by following the not constantly updated gradients (they lead to the correct direction anyway). More generally, given the scale and speed of users in that scenario, maintenance costs are acceptable and do not impact significantly on the application.

In addition, thinking of other application scenarios, it would be possible to programmatically constrain the scope of a tuple (see, e.g., Figure 19) to keep maintenance costs within a reasonable upper bound.

Finally, determining if MetricTuple and SpaceTuple self-maintenance operations are confined is rather easy. In Section 4.2.4, we said that a MetricTuple's maintenance operations are confined to only the node that moves for all the nodes apart from the source, while it spreads across the whole network if the source moves. So MetricTuples must be used carefully, maybe with custom rules in their propagation rules limiting a priori their scope, or by triggering update operations only if the source node moves by at least a certain amount (e.g., a trigger update only if the source moves at least 1 m). The answer for a SpaceTuple, instead, is affirmative since maintenance is strictly locally confined.

It is worth noting that the measures in this section are about the number of operations performed by the tuples. To run the middleware, however, some other operations are required (e.g., the messages sent to probe the neighborhood). These operations are not taken into account in this section since here we wanted to focus on self-maintenance only. However, these are taken into consideration in the results we present in Section 6.3.

6.2.2 *Time.* Let us now focus on the related issue of verifying the time required by the network to fix the tuple fields after some events occurr.

Considering again the tuples in the hierarchy, StructureTuple and MessageTuple tuples are not maintained. SpaceTuple tuples are maintained with only one-hop-bounded operations, so they always maintain with a delay equals to $1T_u$. MetricTuple tuples either maintain with one-hop-bounded operations, and so with a delay of $1T_u$, or, if the source moves, are repropagated and thus, in this case, timing evaluation falls in the previous subsection case.

Then let us focus on HopTuple tuples only. We consider the same experimental setup of the previous subsection. In this set of experiments, however, instead of counting the number of operations required to fix the tuple, we measured the time taken. In particular, for a given network reconfiguration (caused by moving nodes), we recorded the time at which a node performed the last operation to fix the tuple. These times were grouped by the hop distance from the moving node and averaged together. These operations were repeated 100 times and all the outcomes averaged together to obtain the results depicted in Figures 30 and 31. In the former set of experiments, randomly selected nodes moved for a distance equal to one wireless radius (see Figure 30(a)) and a quarter wireless radius (see Figure 30(b)). In the latter set of experiments, the source node moved for a

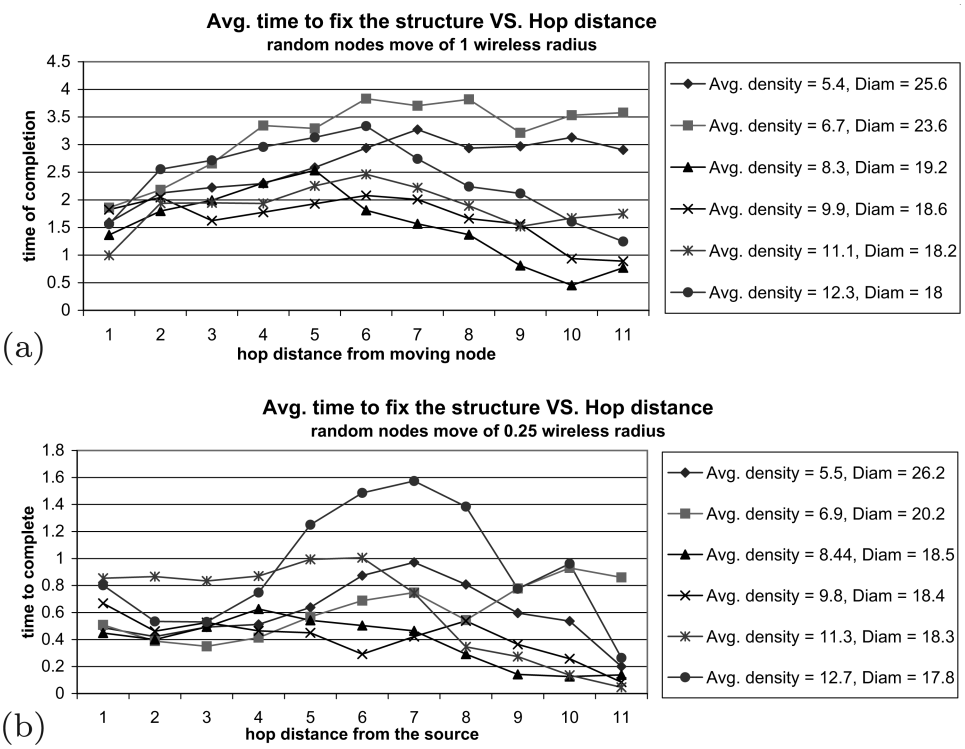


Fig. 30. The time required to fix a HotTuple tuple. (a) Randomly selected nodes move of one wireless radius. (b) Randomly selected nodes move of a quarter wireless radius.

distance equals to one wireless radius (see Figure 31(a)) and a quarter wireless radius (see Figure 31(b)).

In all the experiments reported in Figures 30 and 31, it is possible to see that the time to complete maintenance operations has an increasing then flat behavior. It has a low value near the topology problem. This is because the tuples close to the the topology change are the first to be maintained (so they complete maintenance in a short time). Then it increases with the hop distance, since it requires time for the local algorithm to propagate information across the network to delete and update those tuples that must be maintained. Even further, it stabilizes. This is because maintenance tends to remain confined near the network topology change. This implies that farther nodes do not even take part in the maintenance process (they do not even perceive that the tuple has been maintained). In our experiment, these node were counted as performing maintenance in zero time.

This, of course, depends on the efficacy of the maintenance algorithm, as reported in Figure 28. The more the network is dense, the more is likely that the number of operations will drop to zero and the time to fix the tuple will flatten out.

Also in these experiments, the variance was rather high at 70–80% of the mean for the experiment reported in Figure 30, it was lower at 30–40% of

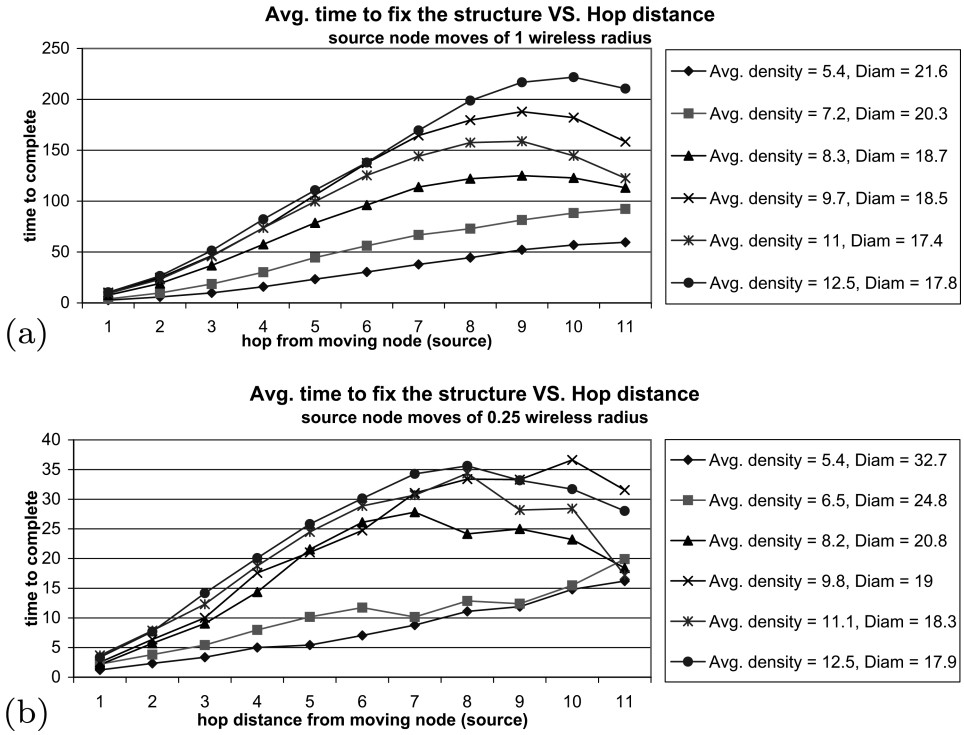


Fig. 31. The time required to fix a HotTuple tuple in the worst-case. (a) The source node moves of one wireless radius. (b) The source node move of a quarter wireless radius.

the mean for the experiment reported in Figure 31. This is in line with the experiments reported in Section 6.2.1, and reflects the fact that reconfigurations are highly topology dependent. Some changes completely destroy the tuple field, while other's do not change it at all.

Looking at these results, we can make the following comments:

—When a random node moves (see Figure 30), the delay in maintaining the tuple field, in the worst case (top of the bell-shaped plot), is close to the delay that would be needed to propagate the tuple there. However, using the self-maintenance algorithm, operations are localized, so the time needed to maintain the tuple tends to drop as the distance from the topology change increases.

—When the source node that injected the tuple moves (see Figure 31), some transient instabilities can arise that incur big delays in tuple maintenance. In particular, whenever a source node move faster than the time required to fix the tuple shape, the maintenance operations temporarily fail—topology reconfigurations happen faster than the time taken to fix them and the tuple field remains unstable. However, from the practical viewpoint, this only causes some loss of accuracy in the perception of the unstable tuple field by other nodes. In any case, when the source stops or simply slows

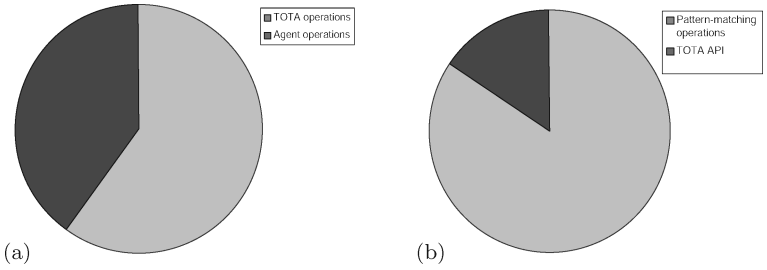


Fig. 32. (a) Internal agent operations versus TOTA operations. (b) API operations versus pattern matching.

down, the self-maintenance algorithms are in any case able to recover the correct tuple configuration.

6.3 Accounting

In every peer-to-peer system relying on cooperation among the nodes of the network, the computational and communication load on a node have to account both for the load introduced by the processes on that node and for that introduced to support other peers' operations (e.g., forwarding other peers' messages). In TOTA, this implies that the load on a node is caused by both the execution of the local agents' operations and by the execution of the operations required to propagate and maintain distributed tuples.

To evaluate the impact on a node of the burden of propagating and maintaining tuples on behalf of other nodes, we performed experiments in challenge scenarios. In particular, and to evaluate the impact on a real node, we exploited our TOTA emulator in the "mixed-mode": TOTA running on a real PDA was virtually embedded in a large (200 nodes) network of simulated TOTA nodes, a portion of which (10%) was mobile. Then, to challenge the scenario, we installed an agent on every node of the network (including the real PDA), each executing a TOTA-intensive application. Each agent continuously and alternatively injected a HopTuple, read another tuple, and deleted the previously injected tuple.

We examined the trace of operations taking place on the PDA with the profiling tool JProfiler[5] and averaged different traces. On the basis of these traces, we counted how many operations were initiated by the agent on the node and how many by neighbor nodes. The resulting ratio is shown in Figure 32(a). There it appears that even in such a challenge scenario only a bit more than half of the costs on a node are due to supporting other nodes' activities. In the vast majority of practical scenarios (e.g., in the case study), where a more limited number of tuples possibly with a confined propagation scope are injected by nodes and less frequently, the burden on a node would be much more limited.

It is also interesting to analyze by what actual activities the above costs are induced. Looking at Figure 32(b), one can see that the costs of accessing the local tuple space to perform pattern matching (which takes place both when the local agents performs read operations, and—most importantly—for managing

[5]`http://www.ej-technologies.com/products/jprofiler/overview.html`.

reactions to events) are the most important ones. Given that the current tuple space implementation (and thus the pattern matching) is implemented in Java and not particularly optimized, this leaves room for further improvements and a reduction of the actual load on TOTA nodes.

6.4 Memory Footprint

A final important consideration relates to the memory footprint of the TOTA middleware and of the tuples involved. This is particularly relevant in that each node, other than the middleware itself, also has to store the tuples injected by other nodes in the network.

With regard to this point, it is important to clarify that, while the TOTA package has to be installed on every node of the network, and thus requires a predefined memory space, the number of tuples being used is completely application dependent. This means that different applications will use different kind of tuples ranging from those in the class hierarchy (having a small memory footprint) to others possibly storing large data sets and thus requiring a lot of memory.

—The memory occupancy of the bare TOTA middleware is slightly less than 4 KB.

—The memory occupancy of the whole tuple class hierarchy presented in Figure 12 is 22 KB and it should be loaded on every TOTA distribution.

—The memory occupancy of the tuples we developed for our experiments (subclassing the tuple hierarchy) was on average just 1.1 KB.

—The memory occupancy of the individual tuple instances is completely dependent on the internal tuple content; thus it is completely application specific.

These measures suggests that TOTA is lightweight enough to be hosted on even small devices such as microsensors. Especially in consideration of the fact that the current implementation is fully Java based and unoptimized, even better results can be expected for an optimized C implementation.

Apart from that, it would be important to implement a sort of garbage collector subsystem to remove unused and irrelevant tuples. This is, however, a general and well-known problem of all tuple space approaches and not peculiar to TOTA.

7. RELATED WORK

A number of recent proposals have addressed the problem of defining supporting environments for the development of adaptive, dynamic, and context-aware applications, suitable for pervasive and mobile computing.

Smart Messages [Borcea et al. 2002; Riva et al. 2007] is an architecture for computation and communication in large networks of embedded systems. Communication is realized by sending “smart messages” in the network, that is, messages which include code to be executed at each hop in the network path. Following this approach, smart messages can encode both routing information and application tasks to be executed on remote nodes (like in the mobile agents

paradigm). Smart Messages shares with TOTA the general idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunks of code to differentiate their behavior as they propagate. The main difference is that, in Smart Messages, messages tend to be used as sort of light-weight mobile agents, in charge of roaming across the network to perform specific tasks. In TOTA, tuples are used to create tuple fields to support context awareness and coordination at the application level.

The L2imbo model, proposed in Davies et al. [2001], is based on the notion of distributed tuple spaces augmented with processes (bridging agents) in charge of moving tuples from 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 TOTA are that, in L2imbo, a tuple is an individual piece of data and its propagation is mainly performed to let them be accessible from multiple tuple spaces. In TOTA, tuples form *fields* and their "meaning" is in the whole tuple field rather than in a single tuple. Because of this conceptual difference, in TOTA tuples' propagation is defined for every single tuple, while in L2imbo it is defined for the whole tuple space.

EgoSpaces [Julien and Roman 2006] is a middleware explicitly targeted at facilitating the acquisition of contextual information in dynamic scenarios. There each node in the network can specify an interest for some contextual information, together with the scope of that interest (e.g., all gas stations within 10 ml). The middleware is then in charge of building a distributed data structure spanning all the peers within the scope of interest to route back the data [Roman et al. 2002]. The main difference between EgoSpaces and TOTA is that in EgoSpaces a node creates distributed data structures mainly to collect information. Accordingly, EgoSpaces data structures represent the sort of queries that spread across the network to get data. In other words, EgoSpaces provides a sort of pull-based interaction mechanism—each agent specifies its egocentric view of the network to pull information. TOTA, instead, can also promote a push-based interaction mechanism. TOTA tuples are spread in the network to diffuse some relevant data and contextual information. In the case study, the application to discover art pieces' information (Section 3.2.1) illustrates both push-based mechanisms (in the first solution) and pull-based mechanisms (in the second solution).

The ObjectPlaces middleware [Schelfthout et al. 2005] shares similar goals with TOTA, although adopting a different architectural approach. ObjectPlaces, at its base, relies on an architecture of nearly independent tuple spaces, in which tuples have the form of objects and can be locally accessed via object matching. However, and this is where the similarity to TOTA comes in, ObjectPlaces enables associating to any object stored in a local tuple space a sort of distributed data structure, called *view*. Such a distributed data structure propagates around the object in the network and defines a sort of field enabling one to detect where an object is, that is, in which direction and at what distance in the network. An important difference between TOTA and ObjectPlaces is that in TOTA the "sender of a message" (i.e., the agent that injects a tuple) determines both who will receive the tuple and what the tuple's content—both are

encoded into the TOTA tuple definition. In ObjectPlaces, instead, the "receiver of a message" (i.e., the agent that builds a view) is the one responsible for deciding both the content and the representation of the messages. ObjectPlaces creates distributed data structures to route information back to the "receiver" node.

Several recent proposals in the area of sensor network programming, such as Hood [Whitehouse et al. 2004], Region Streams [Newton and Welsh 2004], and Logical Neighborhoods [Mottola and Picco 2006], focus on the issue of accessing data in a set of distributed sensors in a flexible and compact way. The common understanding, for all these proposals, is that it should be possible, from the application level, to identify sensing regions of interest and, in such regions, to easily access data in an aggregated way, without having to deal with individual sensor measurements and with network-level data collection issues. To this end, these systems make available APIs and algorithms to dynamically program a sensor network and to enable applications to access aggregated data ignoring low-level algorithmic issues. Although serving different purposes, such proposals share with TOTA the idea of "injecting" into a network the code needed to enable application-specific views of the environment/context. The integration of TOTA (enabling context-dependent perception of environmental characteristics) and such systems (enabling compact perception of aggregated environmental situations) could be very fruitful.

SwarmLinda [Menezes and Tolksdorf 2003] is a middleware applying ant-inspired algorithms to organize and manage distributed tuple spaces. In SwarmLinda, overlay data structures—modeling ants' pheromones—create paths connecting tuple spaces that share similar tuples, thus enabling, for example, the execution of effective pattern-matching operations in the network of tuple spaces. The main difference between SwarmLinda and TOTA is that the former uses the overlay data structure (i.e., pheromones) as an internal mechanism of the middleware. From the programmer point of view, SwarmLinda acts as a single and globally accessible tuple space. Instead, TOTA exposes tuple fields to the programmer as a mechanism to flexibly support application-level coordination.

Another area of related work involves algorithms and mechanisms to route information in dynamic ad hoc networks. Directed diffusion [Intanagonwiwat et al. 2003], content-based multicast [Zhou and Singh 2000], and—more generally—MANET routing protocols [Basagni et al. 2004] are examples of these algorithms. All these systems build field-like data structures (e.g., spanning trees) across the network to support the routing of messages. The main difference between such approaches and TOTA is that these systems are basically routing infrastructures. Instead, TOTA is a run-time middleware that can store information in tuple spaces, can propagate distributed tuples, and provides API and classes to support the creation of TOTA tuples. TOTA is thus more oriented to the application level than the networking level. However, in our future work we plan to better investigate the integration of such algorithms with TOTA.

Finally, an area in which the problem of achieving effective context awareness and adaptive coordination has been addressed via an approach similar

to that of TOTA is Amorphous Computing [Nagpal 2002; Beal and Bachrach 2006]. The particles constituting an amorphous computer have the basic capabilities of propagating and sensing sorts of field-like distributed data structures (similar to TOTA tuples). In particular, particles can transfer an activity state toward directions described by the gradients of these data structures, so as to make coordinated patterns of activities emerge in the system independently of the specific structure of the network. Such mechanism can be used, among other possibilities, to drive particles' movements and to let the amorphous computer self-assemble in a specific shape [Stoy and Nagpal 2004], as well as to orchestrate the activities in groups of mobile robots [Shen et al. 2004]. Although focused on very specific application scenarios, all these approaches share with TOTA the idea of using distributed field-based data structures to drive agent actions.

8. CONCLUSIONS AND FUTURE WORK

TOTA promotes programming pervasive and mobile applications by relying on distributed tuples spread over a network, to be used by application agents both to extract contextual information and to coordinate with each other. As we have tried to show in this article, also with the help of application examples, TOTA suits the needs of modern pervasive and mobile computing scenarios in that (i) distributed tuple structures enable representing contextual information in an expressive, yet simple to be gathered, way; (ii) dynamic and adaptive coordination patterns can be easily enforced in a structured and modular way; (iii) the TOTA middleware, while being light-weight, can effectively support network dynamics by automatically reshaping tuple fields according to the dynamics of the network and of applications.

At the same time, we are aware of a number of current limitations of TOTA that may somewhat limit its degree of applicability and that require further studies. First, proper access control models must be defined to rule access to distributed tuples and their updates, and to protect the privacy of data and of actions. Indeed, security and privacy are challenging issues for the whole area of pervasive and mobile computing, and we are confident several emerging proposals in the area of mobile ad hoc networks [Mishra and Nadkarni 2003] and pervasive computing [Robinson et al. 2005] will be easily portable to TOTA. Second, we think it would be necessary to enrich TOTA with mechanisms and policies to compose tuples with each other, so as to enable the expression of unified distributed data structures from the emanation of multiple sources. This may be needed to integrate correlated information (e.g., global load information [Montresor et al. 2005]), avoid the spreading of an excessive number of independent tuples in the systems, and enable agents to gather with a single read operation more information than currently available in individual TOTA tuples [Mottola and Picco 2006]. Third, there is a lack of general methodologies enabling engineers both to map their application-specific coordination problems into a proper definition of tuples and of their distributed shapes and to properly model/predict the overall behavior of the system under network and environmental dynamics. The increasing

availability of systematic studies in the area of biologically and physically inspired distributed computing [Babaoglu et al. 2006; Zambonelli et al. 2005], in which the exploited coordination models are mostly based on the diffusion of virtual chemical gradients or virtual force fields (and thus directly correspond to the model enforced by TOTA), will help pave the way for the identification of such methodologies.

REFERENCES

Abdelzaher, T., Anokwa, Y., Boda, P., Burke, J., Estrin, D., Guibas, L., Kansal, A., Madden, S., and Reich, J. 2007. Mobiscopes for human spaces. *IEEE Pervas. Comput. 6*, 2, 20–29.

Babaoglu, O., Canright, G., Deutsch, A., Caro, G. D., Ducatelle, F., Gambardella, L., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., and Urnes, T. 2006. Design patterns from biology for distributed computing. *ACM Trans. Auton. Adap. Syst. 1*, 1, 26–66.

Baldoni, R., Marchetti, C., Virgillito, A., and Vitenberg, R. 2005. Content-based publish-subscribe over structured overlay networks. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos.

Basagni, S., Conti, M., Giordano, S., and Stojmenovic, I. 2004. *Mobile Ad Hoc Networking*. Wiley-IEEE Press, Piscataway.

Beal, J. and Bachrach, J. 2006. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intel. Syst. 21*, 2, 10–19.

Bonabeau, E., Dorigo, M., and Theraulaz, G. 1999. *Swarm Intelligence. From Natural to Artificial Systems*. Oxford University Press, Oxford, U.K.

Borcea, C., Iyer, D., Kang, P., Saxena, A., and Iftode, L. 2002. Cooperative computing for distributed embedded systems. In *Proceedings of the International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos.

Cabri, G., Ferrari, L., Leonardi, L., Mamei, M., and Zambonelli, F. 2005. Uncoupling coordination: Tuple-based models for mobility. In *Mobile Middleware*. Taylor and Francis CRC Press, London, U.K., 229–256.

Cabri, G., Leonardi, L., Mamei, M., and Zambonelli, F. 2003. Location-dependent services for mobile users. *IEEE Trans. Syst. Man, Cybernet.—Part A: Syst. Hum. 33*, 6, 667–681.

Camurri, M., Mamei, M., and Zambonelli, F. 2006. Urban traffic control with co-fields. In *International Workshop on Environments for Multi-Agent Systems*. Lecture Notes in Computer Science, vol. 4885. Springer Verlag, Berlin, Germany.

Castelli, G., Rosi, A., Mamei, M., and Zambonelli, F. 2007. A simple model and infrastructure for context-aware browsing of the world. In *Proceedings of the International Conference on Pervasive Computing and Communications*. IEEE Computer Society Press, Los Alamitos.

Cugola, G. and Picco, G. 2006. Reds: A reconfigurable dispatching system. In *Proceedings of the International Workshop on Software Engineering and Middleware*, (Portland).

Davies, N., Cheverst, K., Mitchell, K., and Efrat, A. 2001. Using and determining location in a context-sensitive tour guide. *IEEE Comput. 34*, 8, 35–41.

Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A. 2003. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2, 114–131.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1999. *Design Patterns*. Addison-Wesley, Reading.

Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., and Wicker, S. 2002. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Tech. rep. UCLA/CSD-TR 02-0013. University of California, Los Angeles.

Gelernter, D. and Carriero, N. 1992. Coordination languages and their significance. *Commun. ACM 35*, 2, 96–107.

Hightower, J. and Borriello, G. 2001. Location systems for ubiquitous computing. *IEEE Comput. 34*, 8, 57–66.

Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., and Silva, F. 2003. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw. 11*, 1, 2–16.

JOHANSON, B. AND FOX, A. 2002. The event heap: A coordination infrastructure for interactive work spaces. In *Proceedings of the International Workshop on Mobile Computer Systems and Applications*. IEEE Computer Society Press, Los Alamitos.

JULIEN, C. AND ROMAN, C. 2006. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Trans. Softw. Eng. 32*, 5, 281–298.

LEE, H., MIHAILESCU, P., AND SHEPHERDSON, J. 2007. Realizing teamwork in the field: An agent-based approach. *IEEE Pervas. Comput. 6*, 2, 85–92.

MAMEI, M. AND ZAMBONELLI, F. 2004. Motion coordination in the quake 3 arena environment: A field-based approach. In *International Workshop on Environments for Multi-Agent Systems*. Lecture Notes in Computer Science, vol. 3374. Springer Verlag, Berlin, Germany.

MAMEI, M. AND ZAMBONELLI, F. 2006a. Programming modular robots with the tota middleware. In *International Workshop on Engineering Self-Organizing Applications*. Lecture Notes in Computer Science, vol. 4389. Springer Verlag, Berlin, Germany.

MAMEI, M. AND ZAMBONELLI, F. 2006b. Self-maintained overlay data structures for pervasive autonomic services. In *Workshop on Self-Managed Networks, Systems, and Services*. Springer Verlag, Berlin, Germany.

MAMEI, M., ZAMBONELLI, F., AND LEONARDI, L. 2004. Co-fields: A physically inspired approach to distributed motion coordination. *IEEE Pervas. Comput. 3*, 2, 52–61.

MENEZES, R. AND TOLKSDORF, R. 2003. A new approach to scalable linda-systems based on swarms. In *Proceedings of the ACM Symposium on Applied Computer*. ACM Press, New York.

MISHRA, A. AND NADKARNI, K. M. 2003. Security in wireless ad hoc networks.—A survey. In *The Hand book of Ad Hoc Wireless Networks*, M. Ilyas, Ed. CRC Press, Boca Raton, 499–549.

MONTRESOR, A., JELASITY, M., AND BABAOGLU, O. 2005. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst. 23*, 3, 219–252.

MOTTOLA, L. AND PICCO, G. P. 2006. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the International Conference on Integrated Internet Ad Hoc and Sensor Networks*. ACM Press, New York, 8.

NAGPAL, R. 2002. Programmable self-assembly using biologically-inspired multi-agent control. In *Proceedings of the Intenational Conference on Autonomous Agents and Multi-Agent Systems*. ACM Press, New York.

NEWTON, R. AND WELSH, M. 2004. Region streams: Functional macro-programming for sensor networks. In *Proceedings of the International Workshop on Data Management for Sensor Networks*. ACM Press, New York, 78–87.

O'GRADY, R., GROSS, R., MONDADA, F., BONANI, M., AND DORIGO, M. 2005. Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In *Proceedings of the European Conference on Artificial Life*, (Canterbury, U.K).

PARUNAK, H. V. 1997. Go to the ant: Engineering principles from natural multi-agent systems. *Ann. Operat. Res. 75*, 69–101.

PICCO, G., MURPHY, A., AND ROMAN, G. 2006. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Engin. Methodol. 15*, 3, 279–328.

PISTER, K. 2000. On the limits and applicability of mems technology. In Defense Science Study Group Report. Institute for Defense Analysis. Alexandria.

RIVA, O., NADEEM, T., BORCEA, C., AND IFTODE, L. 2007. Context-aware migratory services in ad hoc networks. *IEEE Trans. Mobile Computing*.

ROBINSON, P., VOGT, H., AND WAGEALLA, W. 2005. *Privacy, Security and Trust Within the Context of Pervasive Computing*. Springer Verlag, Berlin, Germany.

ROMAN, G., JULIEN, C., AND HUANG, Q. 2002. Network abstractions for context-aware mobile computing. In *Proceedings of the International Conference on Software Engineering*. ACM Press, New York.

SCHELFTHOUT, K., HOLVOET, T., AND BERBERS, Y. 2005. Views: Customizable abstractions for context-aware applications in manets. In *International Workshop on Software Engineering for Large Multi-Agent Systems*. Springer Verlag, Berlin, Germany.

SHEN, W.-M., WILL, P. M., GALSTYAN, A., AND CHUONG, C.-M. 2004. Hormone-inspired self-organization and distributed control of robotic swarms. *Auton. Robots 17*, 1, 93–105.

STOY, K. AND NAGPAL, R. 2004. Self-reconfiguration using directed growth. In *Proceedings of the International Symposium on Distributed Autonomous Robotic Systems*, (Toulouse, France).

WANT, R. AND PERING, T. 2005. System challenges for ubiquitous and pervasive computing. In *Proceedings of the International Conference on Software Engineering*. ACM Press, New York.

WEYNS, D., SCHELFTHOUT, K., HOLVOET, T., AND LEFEVER, T. 2005. Decentralized control of EGV transportation systems. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems – Industry Track*. ACM Press, New York.

WHITEHOUSE, K., SHARP, C., BREWER, E., AND CULLER, D. 2004. Hood: A neighborhood abstraction for sensor network. In *Proceedings of the International Conference on Mobile Systems, Application, and Services* (Boston).

ZAMBONELLI, F., GLEIZES, M., MAMEI, M., AND TOLKSDORF, R. 2005. Spray computers: Explorations in self-organization. *J. Perv. Mobile Comput. 1*, 1, 1–20.

ZHOU, H. AND SINGH, S. 2000. Content-based multi-cast (CBM) in ad hoc networks. In *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing*. ACM Press, New York.

Received May 2006; revised September 2007, February 2008; accepted February 2008