Programming Pervasive and Mobile Computing Applications: the TOTA Approach

Marco Mamei and Franco Zambonelli

January 31, 2008

Dipartimento di Scienze e Metodi dell'Ingegneria,
Università di Modena e Reggio Emilia,
Via G. Amendola 2 – 42100 Reggio Emilia, Italy.
marco.mamei@unimore.it, franco.zambonelli@unimore.it

Abstract

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 paper 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 achievement of complex coordination tasks in a fully distributed and adaptive way, mostly freeing programmers and system managers form the need to take care of low-level issues related to network dynamics. This paper includes both application examples to clarify concepts and performance figures to show the feasibility of the approach.

Keywords: Pervasive Computing, Mobile Computing, Coordination, Middleware, Tuple Spaces, Self-adaptation, Self-Organization.

Regular Submission to: ACM Transactions on Software Engineering and Methodology

Previous Publications: A preliminary version of this paper has been published by IEEE CS Press in the Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communications, March 2004. In addition, specific aspects of the TOTA approach have been presented at several focused workshops. This paper extends all of the above with an updated model for tuples, extensive performance evaluations, and more application examples.

Corresponding Author: Marco Mamei (marco.mamei@unimore.it)

1 Introduction

Computing is becoming intrinsically pervasive and mobile [46, 22]. Computer-based systems are going to be embedded in all our everyday objects and in our everyday environments. These systems will be typically communication enabled, and capable of interacting with each other in the context of complex distributed applications, e.g., to support our cooperative activities [4], to monitor and control our environments [16], and to improve our interactions with the physical world [31]. Also, since most of the embeddings will be intrinsically mobile, as a car or a human, distributed software processes and components (from now on, we adopt the term "agents" to generically indicate the active components of a distributed application) will have to effectively interact with each other and orchestrate their activities despite the network and environmental dynamics induced by mobility.

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

Unfortunately, current practice in distributed software development, as supported by currently available middleware infrastructures, is unlikely to effectively address the above 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 paper builds on the lessons of uncoupled coordination models like event-based [23] and tuple-based ones [18], 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, tuples can be "injected" into the network from any node

and can propagate and diffuse across the network accordingly to tuple-specific propagation patterns. The middleware takes care of propagating the tuples and of adapting their distributed shape 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 plan and execute their context-aware activities.

As we will try to show in this paper, TOTA promotes a simple yet flexible approach to programming distributed applications. The approach is suitable for a variety of applications 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 performances and limited overheads. In addition, the fact that TOTA distributed tuples can be considered as sorts of virtual force fields or as sorts of chemical gradients (adopting a natural metaphor) enables to directly enforce in TOTA those phenomena of biological and physical self-organization that are increasingly finding useful applications in modern distributed systems scenarios, such as ant foraging, flocking, self-aggregation, etc. [38, 15, 11].

The remainder of this paper is organized as follows. Section 2 motivates our work, by introducing a case study scenario (that will also act as a working example thorough the paper) and by discussing the inadequacy of traditional approaches to pervasive and mobile computing programming. Section 3 introduces the TOTA approach, by discussing 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 evaluates TOTA in terms of performances and overheads. Section 6 discussed related work. Section 7 concludes 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 inadequacy of traditional approaches in this context.

2.1 Case Study Scenario

Let us consider a big museum, and a variety of tourists moving within it. We assume that each of them is provided with a wireless-enabled computer assistant (e.g., a PDA 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, e.g., properly orienting themselves in the museum, finding specific pieces of arts, or coordinating their activities and movements with other tourists in a group.

In any case, whatever specific service has to 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. 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 in 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 from a museum to a campus or a city, it is also worth noting that traffic management systems [30], warehouse management systems [47], mobile and self-assembling robots [44, 37], sensor networks [22], exhibit very similar characteristics and issues. Therefore, also all our considerations are of a more general validity, besides the addressed case study.

2.2 Inadequacy of Traditional Approaches

Most coordination models and middleware used so far in the development of distributed applications appear inadequate in supporting coordination activities in pervasive computing scenarios. Current proposals can be roughly fitted in 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, e.g., via message-passing or in a client-server way. Modern message-oriented middleware systems like, e.g., Jini [7] and most agent-oriented frameworks (e.g., FIPA-compliant

ones [14]), support such a direct communication model. One problem of this approach is that agents, by having to interact directly with each other, can hardly sustain the openness and dynamics of pervasive computing scenarios: explicit and expensive discovery of communication partners – typically supported by some sort of directory services – has to be enforced. Also, agents are typically placed in a "void" space: the model, per se, does not provide any contextual information, agents can only perceive and interact with (or request services to) other agents, without any higher contextual abstraction. In the case study scenario, to get information about other tourists or art pieces, tourists have to somehow lookup them 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 non-adaptable – for a contingent coordination problem.

Shared data-space models exploit localized data structures in order to let agents gather information and interact and coordinate with each other. These data structures can be hosted in some centralized data-space (e.g., tuple space), as in EventHeap [26], or they can be fully distributed over the nodes of the network, as in MARS [18] and Lime [20]. 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 nearby them and get information about. However, to locate a farther art piece, they should query either a centralized tuple space or a multiplicity of local tuple spaces, and still they would have to internally merge all the information to compute the best route to the target. Similarly, 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 of negotiating with each other to orchestrate their movements. In other words, despite the availability of some local contextual information, a lot of explicit communication and computational work is still required to the application agents to effectively achieve their tasks.

In event-based publish/subscribe models, a distributed application is modeled by a set of agents interacting with each other by generating events and by reacting to events of interest [23]. Typical infrastructures rooted on this model include Siena [19], Jini Distributed Events [7], and recent proposals based on structured overlays [12]. 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 would be to have each tourist notify its 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 eliminate the need for them to explicitly coordinate with each other to orchestrate their movements.

From our viewpoint, the common general problem of the above interaction models is that they are typically used to gather/communicate only a general purpose, not expressive, description of the 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 in a more expressive and possibly application-specific way, agents would be much more facilitated in adaptively deciding 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 achieve the task by exploiting such an 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 [18] and event-based ones [19, 23] and extends them to provide agents with a unified and flexible mechanism to deal with both context representation and agents' interaction.

In TOTA, we propose relying on distributed tuples both for representing application-specific contextual information and for enabling uncoupled interaction among distributed application agents. Unlike traditional shared data space models, tuples are not associated to a specific node (or to a specific data space) of the network. Instead, tuples are injected in the network and can autonomously propagate and diffuse in the network accordingly to specified patterns. Thus, TOTA tuples form sorts of spatially distributed data structures able to act not only as messages to be transmitted between application components but, more generally, as distributed contextual information.

To support this idea, TOTA is composed 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 neighboring 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 nodes' mobility or to nodes' insertions or failures.

The specific nature of the network scenario determines how each node can find its neighbors. In a network scenario without long-range routing protocols being provided or accounted for, it is rather easy to identify the node's neighborhood with the network local topology. For example, in a "bare" mobile ad-hoc wireless network, the neighborhood is identified by the nodes within the range of the wireless link. In this case, which is the one of more interest to this paper, TOTA has to recognize events at the network level to keep an up-to-date list of the in-range nodes. In a network scenario with an established long-range routing protocol (e.g., an Internet overlay), the definition of the node's neighborhood is less trivial. We can imagine however that in such case the term is not related to the real reachability of a node (routing masks network topology), but rather on its addressability (a node can communicate directly with another only if it knows the other node's address). In this case, TOTA can either download from a well-known server the list of addresses representing its neighbors or it can start an expanding ring search to detect close nodes [2].

Upon the distributed space identified by the dynamic network of TOTA nodes, each agent on a node is capable of locally producing tuples and letting them diffuse through the network (see Fig. 1). Tuples are injected in the system from a node, spread hop-by-hop according to tuple-specific propagation rules, and can be locally accessed by other agents on other nodes in an associative way (i.e., via pattern-matching on tuples).

TOTA distributed tuples are characterized by a content **C**, a propagation rule **P**, and a maintenance rule **M**:

T=(C,P,M)

The content **C** is an ordered set of typed fields representing the information carried on by the tuple, and upon which an agent can rely to access tuples via pattern-matching.

The propagation rule **P** determines how the tuple should be distributed and propagated across the network. This includes determining the "scope" of the tuple (i.e., the distance at which such tuple should be propagated and possibly the spatial direction of propagation) and how such propagation can be affected by the presence or the absence of other tuples in the system. In addition, the propagation rules can determine how the tuple's content **C** should change. Tuples are not necessarily distributed replicas: by assuming different values in different nodes, tuples can be effectively used to build a distributed overlay data

Figure 1: (left) The general scenario of TOTA: the environment is a peer-to-peer network; application agents lives 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. (right) Upon changes in the network topology (e.g., the darker node has moved), maintenance rules for tuples can update the tuples' distributed structures to account for the changed conditions.

structure expressing some kind of contextual and spatial information. So, unlike traditional event based models, propagation of tuples is not driven by a publish-subscribe schema, but it is directly encoded in tuples' propagation rule and, unlike events, tuples can change their content during propagation.

The maintenance rule **M** determines how a tuple distributed structure 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 tuples (as specified by the propagation rule) despite network dynamics. To this end, the TOTA middleware supports tuples propagation actively and adaptively: by constantly monitoring the network local topology and the income of new tuples, the middleware automatically re-propagates/re-shapes tuples as soon as appropriate conditions occur. For instance, when new nodes get in touch with a network, TOTA automatically checks the propagation rules of the already stored tuples and eventually propagates the tuples to the new nodes. Similarly, when the topology changes due to nodes' movements, the distributed tuple structure automatically changes to reflect the new topology. 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 "evaporates".

As an example, the tuple shown in Fig. 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 to delete the tuple while it is propagating, in which case the tuple simply represents a volatile event.

3.1 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, e.g., 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 position is implicitly reflected by their access point to the TOTA network. However, this does not exclude the presence of more sophisticated localization tools [25], as discussed in the next subsection.

Moreover, we make the additional assumption that the rough topology of the ad-hoc TOTA reflects the museum topology. This means that there are not network links between physical barriers (like walls). To achieve this property, we can assume that either the devices are able to detect and drop those network links crossing physical barriers (e.g., relying on signal strength attenuation or some other sensor installed on the device). Alternatively, the property can be achieved by assuming that the museum building is pre-installed 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 to 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 problems: *(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 together at a suitable location).

With regard to the first problem, 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.

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 increase it at each hop, can represent the distance of the tuple from its source (i.e., from the art piece itself, see Fig. 2 "Art-Piece Tuple"). 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 backwards (i.e., by following downhill the gradient of the "distance" field), it 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.

Alternatively, we could consider that art pieces do not propagate *a priori* any tuple, but they can sense the arrival of sorts of query tuples propagated by tourists – describing in their content the art piece 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 Fig. 2 "Query Tuple" and "Answer Tuple", where Answer Tuple can propagate following the "distance" field of the associated Query Tuple downhill. It is worth noting, that, since TOTA can maintain (via a proper maintenance rule) the tuple distributed structure coherent despite node movements, the Query Tuple creates a gradient leading to its source even if the source moves. Thus, the answer tuple can reach a tourist even if he is in movement (see Fig. 3(a-b)).

With regard to the motion coordination domain, we focus on a "meeting" service whose aim is to help a group of tourists to dynamically find each other and meet in a room of the museum. Different policies can be though related to how and where a group of tourists should meet. Here we assume that tourists prefer to meet in that room that minimizes the global effort of meeting, that is room representing their "center of gravity". To this purpose, each tourist involved in the meeting can inject and propagate the tuple described in Fig. 2 "Meeting Tuple", to act as a sort of virtual gravitational field of himself. Then, each tourist in the group can start perceiving all such tuples and can start following downhill the resulting global gravitational field (i.e., moving following the gradient of the tuple with the higher "distance" field). This process leads all the tourists to gradually "fall" towards each other until they eventually meet in their center of gravity (see Fig. 3(c)). 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 interesting to notice that, if meeting tuples are properly programmed to maintain their distributed shapes coherent with the actual positions of their sources (i.e., of tourists), then the resulting gravitational fields will be dynamically re-shaped as tourists move, which ensure convergence of the process in general and also in presence of unexpected problems (e.g., a tourists in the group that follows different routes because of crowd in some rooms).

Art-Piece Tuple

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

Query Tuple

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

Figure 2: High-level description of the tuples involved in the museum case study. Art-Piece Tuple is the tuple pro-actively 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 in the meeting application.

(a) (b)

(c)

Figure 3: (a) An agent propagating a Query Tuple to look for a specific information. (b) A suitable art-piece may react to the query by injecting an Answer Tuple that propagates following downhill the Query Tuple. (c) Meeting application: for ease of drawing, the figure shows two agents move toward the leftmost agent that does not move; in general, the three agents would all move toward each other and eventually meet somewhere in between.

3.2 The Concept of Space in TOTA

The type of context-awareness promoted by TOTA is strictly related to spatial-awareness. In fact, the overlay distributed data structures spread in the network in the form of TOTA tuples intrinsically enrich the network with some notion of space.

In the simplest case, a tuple increasing one of the fields of its content as it gets propagated (as the Art-Piece Tuple and the Meeting Tuple in the examples above) defines a sort of "structure of space" representing the network distances from the source. This kind of structures enforce spatial awareness in application agents. In the museum case study, this has been exploited to guide tourists' movements as well as to route messages between tourists and art-pieces according to their current location.

In addition, TOTA also allows dealing with spatial concepts in a much more flexible way. 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. This may be required by several pervasive computing scenarios in which application agents need to interact with and acquire awareness of the physical space and of physical distances.

In order to enforce some physically-grounded concept of space, some kinds of physical localization mechanism must be provided to detect positions of users and of TOTA nodes [25]. These could include GPS-like localization mechanisms to determine the absolute location of an agent, according to some external reference frame [36]; as well as RADAR-like mechanisms to determine the relative location of an agent with respect to other agents. [45].

Once such localization mechanisms are available, 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 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. As another example, one can have a tuple that updates its content (e.g., by increasing an integer value in it) not at each hop but rather each time it has propagated for a certain amount of meters.

Clearly, the kind of localization mechanism available can influences *how* nodes can express and use spatial information. GPS-like mechanism are more suitable at defining "absolute" regions. For example, they allow to easily create tuples that propagate across a region defined by means of the coordinates of its corners (e.g. propagate in the square area defined by (0,0) and (100,100)). RADAR-like mechanism are more suitable at defining "relative" regions, where for example tuples are constrained to travel north form the source or within a specified distance. Further details on these concepts will follow later on.

Other than the network and the physical space, one could think at mapping the nodes

of a TOTA network in any sort of logical or virtual space, and of defining propagation rules and maintenance rules accordingly. On the one hand, if a map is available for a physical region, properly associating physical coordinates with logical places such as streets, buildings, rooms, then one could think at defining tuple propagation rules accordingly to such available logical information (e.g., propagate the tuple only within the current building and increasing the integer value in the content at each new room). On the other hand, if some sort of structured overlay space is defined over a network, such as structured DHTs for P2P computing [41, 10], then one could think of relying on such a virtual space for tuple propagation.

3.3 The TOTA Middleware

From the architectural point of view, the TOTA middleware supporting the above tuple model is constituted by three main parts (see Fig. 4): *(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 about the fact 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 update/re-propagate them accordingly 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 distributed data structure, the TOTA ENGINE needs a mean to uniquely identify tuples in the system. 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 *id*s 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 *id* allows 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 HP IPAQs 36xx equipped with 802.11b wireless card, Familiar LINUX

Figure 4: The TOTA middleware architecture.

[1] and J2ME-CDC (Personal Profile) [5]. IPAQs 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 802.11b 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 [40]) that cannot be provided with sophisticate communication mechanisms. As an additional note, and given the intricacies in accessing network events from Java, the current implementation relies on polling at the TCP level to get events related to network connections and disconnections and to maintain an up-to-date perspective on the TOTA network.

Overall, the memory occupancy of the bare TOTA middleware is slightly less than 4KB of memory. This suggests that TOTA is lightweight enough to be hosted on even small devices such as micro sensors. Especially in consideration of the fact that the current implementation is fully Java-based and un-optimized, and even better results can be expected for an optimized C implementation.

Since we own only 16 IPAQs 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 presence of hundreds of nodes. The emulator, developed in Java, enables examining TOTA behavior in a MANET scenario, in which nodes topology can be rearranged dynamically either by a drag and drop user interface or by autonomous nodes' movements. The strength of our 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 to test applications first in the emulator, then to upload them directly in a network of real devices (see Fig. 5). Moreover, it enables also to run the emulator in a kind of "mixed" mode where

Figure 5: The TOTA emulator. (left) A snap-shot of the emulator. The snap-shot shows the 2D representation of the TOTA network. Moreover, a GUI pops-up when double clicking on a node, e.g., node P39. (right) The same code running on the emulator can be uploaded into the IPAQ, and it can make available the same GUI.

one or more real IPAQs are mapped into specific nodes of the emulator. In this mode, all the IPAQ communication is diverted into the simulation, providing the illusion that the real IPAQ is actually embedded into the simulated network and can communicate with its simulated neighbors. This enables to test interesting scenarios applications in which a user with an IPAQ can interact either with both real network peers and emulated ones, and can pretend being immersed in a very large-scale network.

It is worth outlining that we also managed to integrate the TOTA emulator with a lot of publicly available third-party simulators, enabling us to test the effectiveness of our abstractions in much diverse scenario including robotics [9], urban traffic control [3], and non-player-character control in video-games [6].

4 TOTA Programming

To develop applications upon TOTA, one has to know:

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

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

Figure 6: The TOTA API.

4.1 TOTA API

TOTA is provided with a simple set of primitive operations to interact with the middleware (see code in Fig. 6).

The *inject* primitive is used to inject in the TOTA network the tuple passed as parameter. Once injected, the tuple starts propagating accordingly to its propagation rule, embedded in the definition of the tuple itself, and will automatically maintain its distributed shape according to tuples maintenance rules and transparently to applications.

The *read* primitive accesses the local TOTA tuple space and returns a collection of the tuples locally present in the tuple space and matching the template tuple passed as parameter. A template is a tuple in which some of the content field can be left uninitialized (null).

In addition to this basic reading primitive, 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. The *keyrd* and *keyrdOneHop* methods are analogous of the former two, 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 to evaluate the local shape of a tuple (i.e., for evaluating gradients of specific values in the content of tuples).

The *delete* primitive extracts from the local middleware all the tuples matching the template and returns them to the invoking agent. Here it is important to note that the effect 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 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 opposite *(ii)* deleting it from a different node may have no effect, in that the tuple will be re-propagated there if the maintenance rule specifies so.

The *subscribe* and *unsubscribe* primitives are defined to handle events. These primitives rely on the fact that any event occurring in TOTA (including: arrivals of new tuples, connections and disconnections of peers) can be represented as a tuple. Thus, the *subscribe*

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

Figure 7: This Toy Agent performs three simple actions: it injects a "Hello World" tuple, it looks for another tuple in its tuple space, finally it subscribes to a tuple and react to the income of that tuple, by simply printing out a string.

primitive associates the execution of a reaction method in the agent in response to the occurrence of events matching the template tuple passed as first parameter. Specifically, when a matching event occurs, the middleware invokes on the agent a special *react* method and passes to it, as parameters, the reaction string (to specify which subscription has been triggered) and the matching event. The *unsubscribe* primitive removes all the matching subscriptions. It is worth noting that, despite the fact that all the TOTA read methods are non-blocking, 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.

Fig. 7, coding a toy application agent, can be of use to clarify the above concepts and to show the basic usage of TOTA primitives.

4.2 Programming Distributed Tuples

Other than the TOTA API, a suitable programming model is required to build TOTA distributed tuples. Distributed tuples have been designed by means of objects: the object state models the tuple content, the tuple's propagation has been encoded by means of a specific *propagate* method, while maintenance rule can be implemented by properly coding an additional *react* method. In addition, the *init* method initializes the tuple as it arrives on a node by providing it the reference to the local TOTA middleware. This leads to the definition of the abstract class *TotaTuple*, providing a general framework for programming tuples (see code in Fig. 8).

```
abstract class TotaTuple {
 protected TotaInterface tota;
 /* instance variables represent tuple fields */

 /* this method inits the tuple, by giving a reference
 to the current TOTA middleware */
 public void init(TotaInterface tota) {
 this.tota = tota;
 }
 /* this method codes the tuple actual actions */
 public abstract void propagate();
 /* this method enables the tuple to react
 to happening events */
 public void react(String reaction, String event)
 {
}}
```

Figure 8: The main structure of the TotaTuple class.

We emphasize that TOTA tuples do not own internal threads, but their code – for initialization, propagation, and maintenance – is actually executed by the middleware, i.e., 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.

In general, a programmer can create new tuples by sub-classing the *TotaTuple* class and by specifying its own content, propagation rule, and maintenance rule.

Defining the content simply implies identifying the instance variables. A general constructor method relying on Java reflection enables then to create a tuple by passing it an array of objects that are cast to the correct Java type. This relaxes programmers form the need of defining a constructor (they simply have to invoke that of the superclass) and also enables the creation of template tuples with uninitialized values (which are used in pattern matching).

The correct writing of proper propagation and maintenance rules, instead, may not be trivial. To facilitate this task, TOTA makes available a whole tuples' class hierarchy (see Fig. 9). 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).

The key classes in the hierarchy include: *StructureTuple*, *MessageTuple*, *HopTuple*, *MetricTuple* and *SpaceTuple*. In the following, we are going to describe in detail these classes

Figure 9: Some key classes of the TOTA tuples' class hierarchy.

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

Figure 10: The propagate method in the StructureTuple class

showing how they can be effectively exploited to create application-specific classes.

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 breadth first expanding ring 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 Fig. 10). 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.

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 in any case be of some use in applications where the network infrastructure is relatively static and thus there is not the need to constantly update and maintain the tuple distributed structure. However, what matter in the *StructureTuple* class is that it also facilitate the definition of maintenance rules by specifying how tuples should subscribe to events during propagation in order to later support their own maintenance.

The *StructureTuple* defines the *propagate* method around the following four methods:

decideEnter, *decidePropagate*, *changeTupleContent*, *makeSubscriptions*. In addition, it exploits two private methods of the TOTA API, i.e., *store* and *move*. The procedure for propagation is as follow:

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

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

- The method *changeTupleContent* 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 distributed data structure would ever be formed.

- Then, if the *decidePropagate* method returned true, the tuple is propagated to all the neighbors via the *tota.move(this)* primitive. With this method, the tuple will reach 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.

Programming a TOTA tuple to create a distributed data structure basically reduces inheriting from the above class to overload the four methods specified above to customize the tuple behavior. For instance, different conditions can be evaluated to decide if a propagated tuple should enter a node (*decideEnter*), if it should further propagate (*decidePropagate*),

or if it should change its content and how (*changeTupleContent*). And one can even query the local tuple space to evaluate these conditions. We also emphasize that the presence of both the *decideEnter* and *decidePropagate* methods derive from the fact that the decision of whether a tuple should propagate from a node A to a node B can be sometimes based on conditions known at the node A, 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 B, in which case one can evaluate the condition within the *decideEnter* method.

Let us now present an example of how one can define a new tuple class by inheriting from the basic *StructureTuple* class. The *NotMaintainedGradient* class (see code in Fig. 11) defines tuples that flood the whole network and have an integer hop-counter that is increased by one 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 enter in a node not only if in the node there is not the tuple yet – as in the base implementation – but also if there is the tuple with an higher hop-counter. This allows to enforce the breadth-first propagation assuring that the hop-counter truly reflects the hop distance from the source. We emphasize that one could also think at overloading the *decidePropagate* method to bound the propagation of the tuple and confine it to a limited number of network hops from the source (or, even better, one can subclass from *NotMaintainedGradient* to define such a *BoundedNotMaintainedGradient* class).

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, i.e., 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 not be acceptable 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 not stored in the local tuple spaces but just flow in the network.

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

```
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++;
}}
```

Figure 11: The *NotMaintainedGradient* class defines a tuple that floods the network and have an integer hop-counter that is increased by one at every hop.

```
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,(ReactiveComponent)this,"ERASE");
 }
 public void react(String reaction, String event) {
  if(reaction.equals("ERASE")) {
   tota.delete(this);
}}}
```

Figure 12: The *MessageTuple* class defines a tuple that floods the network with an integer value and delete itself after some time has passed, to act as a sort of message that spread in the network.

after some time passed. The code of the *MessageTuple* class is in Fig. 12. It is worth noticing that TOTA takes care of wrapping low-level data (such as the system clock) into 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 from Fig. 10). 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 "wave-front" has passed. Clearly, setting the optimal time before deletion is not trivial. If the tuple propagates in a breadth first manner in a nearly regular planar network, such time can simply be set to greater than the time required for the tuple "wave-front" to proceed two-hops away. However, if the tuple is intended to propagate in specific directions and/or the network topology has large "holes" leading to circular paths, a short LEASE time can lead to message tuples that endlessly circulate in the network. For this reason, *MessageTuples* should consider worst case LEASE time or, even better, should consider confining the propagation distance of a tuples to ensure that propagation will eventually stop.

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 Smart Messages approach [16]). To make an example, one can define by inheritance from *MessageTuple* a *DownhillTuple* class, defining tuples that propagate 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 Fig. 13).

By combining these tuples with *StructureTuple*, 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 that we already proposed for the case study: a tourist can propagate queries for art pieces in the form of "Query Tuple" defined starting from *StructureTuple*, while art pieces can react to such queries by injecting an "Answer Tuple" defined starting from *DownhillTuple*.

At this point, based on the examples reported so far, it should also start becoming clear how TOTA can be exploited to effectively enforce naturally-inspired phenomena of self-organization. On the one hand, TOTA tuples can be exploited to define sorts of virtual force fields emanating from a source or sorts of diffusing chemical gradients. In the latter case, one could also exploit maintenance rules similar to that defined for the *MessageTuple*

```
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;
}}
```

Figure 13: The *DownhillTuple* class defines tuples that propagate by following downhill the trial left by another tuple. 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*).

class to enforce slow propagation and slow evaporation of tuples, as it occur in real chemical diffusion. On the other hand, TOTA tuples can be defined that propagate by being somewhat "attracted" by the virtual force fields (or chemical gradients) defined by other tuples. These two mechanisms are at the basis of most phenomena of self-organization in natural and physical systems, such as ant-foraging, self-aggregation, flocking in birds [38, 15, 11].

However, to complete the picture, TOTA tuples must be provided with the capability of adaptively maintaining a coherent distributed structure in response to the dynamics of the underlying network environment.

4.2.3 HopTuple

The *HopTuple* class inherits from *StructureTuple* to define distributed data structures that are able to self-maintain their propagated structure to reflect dynamic changes in the network topology (see Fig. 14, as well as Fig. 1), there included changes in the network positioning of the injecting node.

Similarly to the class *NotMaintainedGradient* in Fig. 11, the *HopTuple* defines an integer

(a) (b)

Figure 14: *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 auto-delete to reflect the new network situation.

hop counter as content, and it overloads the *decideEnter* method so as to allow the entrance in a node only if the tuple is not already there and if it does not have a lower hop-count. This allows to enforce the breadth-first propagation assuring that the hop-counter truly reflects the hop distance from the source [28]. Moreover, *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* distributed tuple structure. Given one of its local tuples 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 distributed tuple structure of X; Y is at one-hop distance from X; and the hop-counter value of Y is equal to that of X minus one. With such a definition, a supporting tuple of X is a tuple 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 firstly injected the tuple (i.e. $hopvalue = 0$). The basic idea of the algorithm is that a tuple that is 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 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 sub-network are deleted. On the contrary, in the case 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 distributed tuple structure.

Readers can refer to [29] for a detailed description of the complete algorithm and of its stabilization properties. What we emphasize here is that, as complex as this algorithm and its coding can be, users do not have to worry: they can simply exploit the algorithm by

inheriting from the *HopTuple* class.

The self-maintained tuple structures 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 in the networks to fully disregard the actual structure and dynamics of the underlying network. Once the tuple is injected, its maintenance rules ensure that the distributed tuple structure will always reflect the intended propagation pattern. This moves away from the agent the burden of maintaining the tuple shape 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.

Indeed, all the examples presented above should have better exploited tuple derived from the *HopTuple* class, in order to tolerate network dynamics and dynamic movement of the source.

Let us consider the meeting tuple in the case study (see Fig. 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 express 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 delay in propagation of the tuples and of its 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 [11].

To better clarify the power and expressiveness of such self-maintained tuples, let us discuss another example. A *FlockingTuple* class can be created by inheritance from *HopTuple*, to define distributed data structures in which an integer in the content of the tuples has a minimum at a specific distance (RANGE) from the injecting agent. Coding such a tuple is trivial (see Fig. 16), since one has simply to *(i)* place in the object state an integer representing the flock value and *(ii)* overload the *changeTupleContent* to let the tuple assume the intended shape. Note that the "hop" variable is maintained in the super-class *HopTuple*.

The name of the *FlockingTuple* class comes from the fact it can be employed to realize an interesting application forcing a group of agents to maintain a specific regular formation, mimicking that process of adaptive self-organization that is the flocking of birds. Flocks of birds stay together, coordinate turns, and avoid colliding each other by following a very simple algorithm [15]. Their coordinated behavior can be explained by assuming that each bird tries to maintain a specified separation from the nearest birds and to match nearby birds' velocity. To implement such a coordinated behavior with TOTA, we can have each agent generate a *FlockingTuple* assuming the minimal value at a distance expressing the intended spatial separation between agents. The final shape of the resulting distributed

Figure 15: (left) Ideal shape of the flock tuple. (right) When all the agents follow other agents' tuples they collapse in a regular grid formation.

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

Figure 16: A *FlockingTuple* creates a data structure that has a minimum at a specific distance (RANGE) from the injecting agent.

tuple resembles the function depicted in Fig. 15-left. *FlockingTuples* are always updated to reflect peers' movements. To coordinate movements, peers have simply to locally perceive the generated tuples and follow them downhill. The result is a globally coordinated movement in which peers maintain an almost regular grid formation (see Fig. 15-right). For instance, in the case study, one can think of exploiting this mechanism to orchestrate in a fully distributed way the movememt of security guards while monitoring the museum.

More in general, all natural phenomena of adaptive self-organization relying on the diffusion of some sort of field gradient (chemical, physical, or simply perceptual, as in the case of birds) can be easily mimicked in TOTA by defining self-maintained tuples with specifically conceived content and propagation rules. There, the natural endeavor of physical and chemical fields of continuously reflecting the properties of the surrounding physical space (which contributes making such phenomena adaptive and robust) translates in the capability of self-maintained TOTA tuples of continuously reflecting the structure of the underlying network.

As an additional note, also in the case of *HopTuple* sub-classes 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-maintainance, which continue performing its work in the sub-part of the network in which the tuple is intended to be

Figure 17: *MetricTuple* and *SpaceTuple* tuples create a shared coordinate system, centered in the node that injected the tuple.

propagated.

4.2.4 MetricTuple and SpaceTuple

All the above presented tuple classes rely on propagation patterns directly related to the underlying network structure, i.e., 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 propagation and maintenance of TOTA tuples based on the actual localization of TOTA nodes. In particular, both these tuple classes define three float numbers *(x,y,z)* as a content, representing the physical coordinates in a 3D reference space. Once one of these tuples is injected in the network, the *(x,y,z)* tuple is set to *(0,0,0)*, and the tuple propagates by having its content change so that 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 Fig. 17). In other words, these tuples define a coordinate system centered around the source node.

The current implementation of *MetricTuple* and *SpaceTuple* tuples supports both the presence of GPS-like localization and of RADAR-like localization (see also Subsection 3.2).

The implementation of these tuples in the presence of GPS-like localization is straightforward. Once injected, the tuple will store the injecting node GPS coordinates and will initialize its content to (0,0,0). Upon reaching a new node, it will change the content to the GPS coordinates of the new node translated back by the injecting node coordinates. Tuple

Figure 18: *MetricTuple* and *SpaceTuple* tuples create the shared coordinate system, by having each node change the content of the tuples on the basis of the coordinates provided by the RADAR-like device.

maintenance proceeds similarly: when a node moves (let us for now assume this is not the source node), the tuple locally changes its content by accessing the new GPS information locally available to the node that has moved.

The implementation of *MetricTuple* and *SpaceTuple* tuples, given the availability of a RADAR-like localization, is more complicated. Here the goal is to create a tuple class that combines the local coordinate systems, as made available by RADAR-like devices, into a shared coordinate system, with the center in the node that injected the tuple. To explain how this can be achieved let us consider Fig. 18. The tuple (0,0,0) travels from P1 to P2 and it changes its content there. Specifically, it subtracts from its old value the coordinates of P1 as sensed by the RADAR-like device in P2. Thus $(0-(-100), 0-(-20), 0-0) = (100, 20, 0)$. It is worth noting that, in the figure, all the private coordinate systems are aligned. So combining them is just a matter of adding the coordinates. However, this perfect alignment is unlikely to happen and slightly more complex (geometric) combination will be required. Further propagation hops (from P2 to P3, and form P2 to P4) proceed analogously. Tuple update proceeds similarly: once these tuples have been propagated, if a node moves (and given this is not the source node), only its tuple local value is affected, while all the others are left unchanged. In fact, the others physical positions with respect to the source do not change. Upon a movement *MetricTuple* and *SpaceTuple* tuples read the RADAR localization and adjust their values accordingly (see Fig. 19).

To key difference between *MetricTuple* and *SpaceTuple* tuples relates to what happens when the source moves. For *MetricTuple* tuples, the origin of the coordinate system originated from the injected tuple is anchored to the source node, and when the source moves all the coordinates in the propagated tuple structure will be updated. This of course can lead to performance problems: if the source is highly mobile, this will imply a continuous update

Figure 19: Self-maintenance in *MetricTuple* and *SpaceTuple* tuples. Since the tuple on P3 does not change position with respect to the tuple source, it does not change its content. P2 updates its content on the basis of the new RADAR reading.

of the whole distributed tuple 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 distributed tuple structure is triggered. *SpaceTuple* tuples solves 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.

Let us now show two examples of tuple classes derived from the above: *(i) DistanceTuple* and *(ii) FlockMetricTuple*.

A *DistanceTuple* is a tuple that holds as a variable in its content the spatial distance from the source (which can be easily computer 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 Fig. 20).

The *FlockMetricTuple* tuple class encodes the already described flocking tuple (see Fig. 15-left), and can be used for the same purpose. However, in this case, the minimum of the flock fiend can be set to a specific physical distance, rather than to a network-hop distance (see code in Fig. 21).

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

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

Figure 20: A *DistanceTuple* is a tuple that holds the spatial distance from the source.

```
public class FlockMetricTuple extends MetricTuple {
 private int a = 30;
 public int value = 0;
 protected void changeTupleContent() {
  super.changeTupleContent();
  int d = (int)Math.sqrt((x*x)+(y*y)+(z*z));
  /* d^4 - 2a^2d^2 is a kind of Mexican-hat
  function suitable for flocking */
  value = (d*d*d*d) - 2*(a*a)*(d*d);
}}
```

Figure 21: A *FlockMetricTuple* encodes in its shape the flock-field described in Fig. 15-left.

4.3 Programming Agents

The last step involved in programming TOTA applications is coding the agents that use the TOTA API and that 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.

4.3.1 Gathering Contextual Information

The first approach proposed in Subsection 3.1 considers that art-piece agents (i.e, agents residing on art pieces) propagate in the network a tuple describing themselves, and that tourist agents can simply query the local tuple space to discover the presence of art-piece and their estimated location.

The code for these two agents is in Fig. 22 and in Fig. 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, e.g., 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).

In these examples, the *MyGradient* class inherits from *NotMaintainedGradient*, which does not implement maintenance rules. Alternatively, if one foresee dynamics in the museum

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

Figure 22: *ArtAgent* first solution.

```
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);
}}}
```

Figure 23: *TouristAgent* first solution.

structure or possible movements of the art-piece (due to, e.g., organization of temporary exhibitions), one could have simply defined the *MyGradient* class by extending the *HopTuple* one, to get advantage of its self-maintenance properties.

The same example can be easily translated to work in different scenarios, e.g., for outdoor. 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 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 in outdoor scenarios. In addition, for such outdoor scenario, it can make sense to bound the propagation of art piece tuples to, say, a few kilometers (why should a tourist care of a very far away monument?).

Beside the spreading of rather static information related to art pieces and monuments,

the same example can by applied nearly as it is to spread and retrieve information about any relevant situation currently occurring somewhere in an environment: the presence of policemen, the emergence of a traffic jam, a road show going to begin.

The second approach sketched in Subsection 3.1 considers that art-pieces agents do not propagate tuples, but are instead programmed to sense the income 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 tourist, it has the advantage of limiting the spreading of tuples in the network, and may thus be suggested for the retrieving of non-critical contextual information.

Such second approach is coded by the *ArtAgent2* and the *TouristAgent2* classes presented in Fig. 24 and Fig. 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, i.e., 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 structure of that of *MyGradient*. Such *MyDownhillTuple* propagates by simply following backward the query tuple to reach the tourist agent which issued the request.

A tourist agents, by its side, performs just two simple operations: it injects in 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 ("Monna Lisa"). The associated reaction *displayReaction* is executed on receipt of such tuple to print out the full content of the received event tuple in the user interface.

Also for this second solution, we can consider the use of *MetricTuple* tuples for outdoor. Also, we emphasize that since *MyDownhill* tuples are self-maintained, the answer to a query can reach a tourist even while in movement.

In addition, for both the proposed solutions, it may make sense to adopt temporary tuples, i.e., tuples with a limited time-to-live (as already specified in Subsection 4.2.2 and accordingly to the maintenance rules of Fig. 12), so as to avoid leftovers of no longer meaningful tuples.

4.3.2 Meetings and Motion Coordination

With regard to the meeting application, the algorithm followed by *MeetingAgents* (see code in Fig. 26) is very simple: agents have to determine the farthest peer, and then move (better: suggest their user where to go) by following downhill that peer's presence tuple.

```
public class ArtAgent2 implements AgentInterface {
 private TotaMiddleware tota;
 private Object description[];
 /* agent body */
 public void run() {
  /* subscribe to the query */
  description =  new Object[] {"Monna Lisa", "Da Vinci", "1492", "Room X"}
  MyGradient 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) {
  MyDownhillTuple answer = new MyDownhillTuple(description);
  tota.inject(answer);
}}
```

Figure 24: *ArtAgent* second solution.

```
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);
}}}
```

Figure 25: *TouristAgent* second solution.

In this way agents will meet in their center of gravity (i.e., the 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 in 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 location of other users.

Starting form 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 tourist. For example, a tourist guide could inject a tuple similar to the *MeetingGradient* to be easily reachable by tourists (that could follow the tuple downhill to reach the guide). As another example, the guides themselves could try to follow each 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 crowd within the museum. To this end, we can assume that, for each room, a *Crowd* tuple is spread across the museum with a field whose value represents to the number of people in that room. Following this *Crowd* tuple downhill, tourists could try to stay away form crowd and queues while visiting the museum. Further details on these and other motion coordination examples can be found in [31].

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

5 Experimental Evaluation

The effectiveness of the TOTA approach is of course related to the costs, performances, and overheads, involved in propagating and maintaining TOTA distributed tuples. To this end, we need to:

- analyze and evaluate the costs involved in propagating and deleting tuples across a network;
- evaluate the impact of self-maintenance algorithms on a network, which also strictly affects the scalability of the approach;
- analyze the impact on TOTA nodes of the various operations in which they are involved during the execution of TOTA applications.

```
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);
}}}
```

Figure 26: Agent example: *MeetingAgent*

5.1 Tuple Propagation and Deletion

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

In general, the multi-hop 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 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 form 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 the tuple and have it 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 to letting the stream of data arrive on the other node.

These values as measured in our PDA implementation of TOTA are reported in Fig. 27.

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

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**

Figure 27: Time costs in a PDA.

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 [24]). 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-hop away from the source has been recorded and averaged together over a large set of experiments. The results are depicted in Fig. 28(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 3 hops. Then, we propagated tuples in there measuring delays. The real time to propagate a tuple in the PDA network, averaged over a number of experiments, is reported in Fig. 28(b). In both the experiments, it is easy to see that the number of operations required is slightly greater than the ideal model, though still inherently linear with the number of hops. The same results hold in the case of the tuple deletion process.

To get the real meaning of these results, one can consider that for a network of nodes with a wireless radius of $20m$ (that could be the typical radius of WiFi in a cluttered environment), the physical speed for tuple propagation would be around 75 m/s, i.e., 270 Km/h. Such a speed enables to effectively adopt TOTA in scenarios involving humans, robots, as well as cars in urban traffic, all of which exhibiting 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.

5.2 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

(a)

(b)

Figure 28: (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.

its scalability.

In the following of this subsection, we evaluate self-maintenance algorithms for the different classes of the TOTA hierarchy of Fig. 9, with a specific emphasis on the self-maintenance algorithm of the *HotTuple* class.

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 re-usable in the vast majority of cases. Subclassing from them to implement different self-maintenance algorithms is always possible, but may introduce totally different behaviours and performances.

5.2.1 Overhead and Scalability

Let us now focus on the problem of verifying how the performances and the overhead induced by tuple maintenance algorithms affects the nodes of the network, and whether such performances 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 introduces 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 not feasible to generally adopt such tuples in the presence of dynamic networks. On the contrary, 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 independent of the system size and, thus scalable.

Establishing by analysis how the *HopTuple* self-maintenance algorithm impacts on the network is very complex, also because the actions performed by it strongly depends on the network topology [29] (see Fig. 14). 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 run several simulations varying the node density and their initial position. In particular, we run six sets of experiments where we randomly deployed 200, 250, 300, 350, 400, 450 nodes in the same area; thus obtaining an increasing node average density and a shrinking network diameter. All the experiments were repeated a large number (over 100) of times with different initial network topologies and the results were averaged together. Specifically, we conducted two classes of experiments.

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.

In a first set of experiments, a randomly chosen node injects a *HopTuple* in the network. After that, randomly chosen nodes start moving independently (following a random way-point motion pattern) perturbing the network. In particular, a randomly picked node moves randomly for a distance equals to 1 wireless radius. This movement changes 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, is recorded. Specifically, we evaluate the average number of messages exchanged by nodes located at x-hop away from the moving node. Then, we average these numbers over a large set of topology changes. The results of this experiment are in Fig. 29(a).

The experiments reported in Fig. 29(b) have been conducted in the same manner. This time, however, nodes move for a distance of 1/4 wireless radius. This second set of experiments is intended to show what happens for small topology reconfigurations (wider reconfigurations can be depicted as a chain of these smaller ones).

In a third set of experiments, nodes are not free to move independently from one other.

(a)

(b)

(c)

Figure 29: 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 1 wireless radius. (b) random node movements for 1/4 wireless radius. (c) a group of nodes moving for 1 wireless radius.

They are grouped with some of their neighbors and once a node moves all the nodes belonging to the group move solidally. More specifically, in the reported experiment, a randomly selected node moves for a distance of 1 wireless radius and all the nodes directly connected to it (i.e. in its one-hop neighborhood) move 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 in Fig. 29(c). We repeated this experiment with also larger groups of nodes (i.e when a node is going to move, all the nodes within 2 or even 3 hops move together) always obtaining analogous results.

Looking at these figures, the most important consideration we can make is that, when a node moves and the network topology changes consequently, a lot of update operations are required near the area where the topology changes, while only few operations are required far away from it. This implies that, even if the network and the tuple being propagated

have no well-defined boundaries, the operations to keep their shape consistent are strictly confined within a locality scope. This result is even more significant if compared to the average network diameter (averaged over the various experiments). It is easy, in fact, to see that the number of operations required to maintain a TOTA tuple falls close to zero well before the average diameter of the network, thus confirming the quality of our results. This fact supports the idea that the operations to fix distant concurrent topology changes do not add up, making the approach scalable.

Experiment 2: This array of experiments is similar to the previous one, but this time only the source node that injected the *HopTuple* tuple moves altering the network topology. This is a worst-case scenario, because when the node that injected the tuple (source node) moves, major changes are expected in the data structure shape. In Fig. 30(a) the source node moves randomly of a distance of 1 wireless radius. In Fig. 30(b) it moves of 1/4 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 decreases with the distance from the topology change, but such number remains relevant even at high distances. This implies that 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.

Finally, determining if *MetricTuple* and *SpaceTuple* self-maintenance operations are confined is rather easy. In subsection 4.2.4 we described that a *MetricTuple*'s maintenance operations are confined to the only node that moved 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. trigger update only if the source moves at least 1m). The answer for a *SpaceTuple*, instead, is affirmative since maintenance is strictly locally confined.

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

5.2.2 Time

Let us now focus on the related issue of verifying the time required by the network to fix the structure of distributed tuples after some events occurred.

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

(a)

(b)

Figure 30: 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 1 wireless radius. (b) The source node moves for 1/4 wireless radius.

moves, they are repropagated and thus, in this case, timing evaluation falls in the previous subsection case.

Then, let us focus the attention on *HopTuple* tuples only. We consider the same experimental set up of the previous sub-section. 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 performs the last operation to fix the tuple. These times are grouped by the hop distance from the moving node and averaged together. These operations have been repeated several (more then 100) times and all the outcomes have been averaged together to obtain the results depicted in Fig. 31 and Fig. 32. In the former set of experiments, randomly selected nodes move for a distance equals to 1 wireless radius (see Fig. 31(a)) and 1/4 wireless radius (see Fig. 31(b)). In the latter set of experiments, the source node moves for a distance equals to 1 wireless radius (see Fig. 32(a)) and 1/4 wireless radius (see Fig. 32(b)).

In all the reported experiments in Fig. 31 and Fig. 32 it is possible to see that the time to complete maintenance operations has a rough bell-shaped behavior. It has a low value near the topology problem. This is because, the tuples close the the topology change are the first to be maintained (so they complete maintenance in short time). Then it

(a)

(b)

Figure 31: The time required to fix a *HotTuple* tuple. (a) Randomly selected nodes move of 1 wireless radius. (b) Randomly selected nodes move of 1/4 wireless radius.

(a)

(b)

Figure 32: The time required to fix a *HotTuple* tuple in the worst-case. (a) The source node moves of 1 wireless radius. (b) The source node move of 1/4 wireless radius.

increases with the hop-distance, since it requires time to the local algorithm to propagate information across the network to delete and update those tuples that must be maintained. Even further it decreases. This is because maintenance tend to remain confined near to 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 are counted as performing maintenance in 0 time. This, of course, notably reduces the average time to fix the tuple and thus it creates the descending part of the bell-shaped plot.

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

- When a random node moves (see Fig. 31) the delay in maintaining the tuple, 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 tend to drop close to 0 as the distance from the topology change increases. This confirms the general scalability of tuple maintenance.

- When the source node that injected the tuple moves (see Fig. 32), some transient instabilities can arise that incur in 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 then time taken to fix them and the tuple structure remains unstable. However, from the practical viewpoint, this only causes some loss of accuracy in the perception of the unstable tuple structure by other nodes. In any case, when the source stops or simply slow down, the self-maintenance algorithms are in any case able to recover the correct tuple configuration.

5.3 Accounting

In every peer-to-peer system relying on cooperation among the nodes of the network, the computational and communication load on a node has 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 challenged 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 IPAQ was virtually embedded in a large (200 nodes)

(a) (b)

Figure 33: (a) Internal agent operations vs. TOTA operations. (b) API operations vs. pattern matching

network of simulated TOTA nodes, a portion of which (10%) mobile. Then, to challenge the scenario, we installed an agent on every node of the network (there included the real IPAQ), each of which executing a TOTA-intensive application. In particular, 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 IPAQ with a profiling tool [8] 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 neighbors nodes. The resulting ratio is in Fig. 33(a). There, it appears that even in such a challenged 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 worth interesting to analyze by what actual activities the above costs are induced. Looking at Fig. 33(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 when tuples coming from other nodes are to be propagated) 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 reduction of the actual load on TOTA nodes.

6 Related Work

A number of recent proposals address the problem of defining supporting environments for the development of adaptive, dynamic, context-aware distributed applications, suitable for pervasive and mobile computing, and sharing some characteristics with TOTA.

Smart Messages [16], rooted in the area of active networks, is an architecture for computation and communication in large networks of embedded systems. Communication is realized by sending "smart messages" in the network, i.e., messages which include code to be executed at each hop in the network path. Smart Messages shares with TOTA the general idea of putting intelligence in the network by letting messages (or tuples) execute hop-by-hop small chunk of code to differentiate their behavior as they propagate. The main difference is that, in Smart Messages, messages tend to be used as sorts of light-weight mobile agents, in charge of roaming across the network to perform specific tasks rather than, as in TOTA, to support context-awareness and adaptive coordination at the application level. A recent proposal of the Smart Messages research group, though, goes in that directions by exploiting smart messages to enforce spatial-awareness and adaptive programming of spatially-situated activities [17], in a way similar to that of TOTA.

The L2imbo model, proposed in [21], 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, tuples are conceived as "separate" entities and their propagation is mainly performed to let them being accessible from multiple tuple spaces. In TOTA, tuples form distributed data structure and their "meaning" is in the whole data structure rather than in a single tuple. Because of this conceptual difference, tuples' propagation is defined for every single tuple in TOTA, while is defined for the whole tuple space in L2imbo.

Lime [39] exploits transiently shared tuple spaces as the basis for interaction in dynamic network scenarios. Each mobile device, as well as each network node, owns a private tuple space. Upon connection with other devices or with network nodes, the privately owned tuple spaces can merge in a federated tuple space, to be used as a common data space to exchange information. TOTA defines a more flexible model than that of Lime. It is possible, via specific propagation rules, to have tuples distributed only in a local neighborhood, so as to achieve the same functionalities of the locally shared tuple spaces of Lime. In addition, propagation rules enable more elaborated forms of information sharing other than simple local merging of information. Similar considerations may apply with regard to other proposals for shared distributed data structures (e.g., the XMIDDLE middleware [32]), as well as to the recent extension of Lime for sensor networks [20].

EgoSpaces [27] builds as a sort of additional middleware layer over Lime. It is explicitly targeted at facilitating the acquisition of contextual information in dynamic scenarios (i.e., MANETs). 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 miles).

The middleware is then in charge to build a distributed data structure (i.e., a shortest path tree) spanning all the peers within the scope of the interest. This data structure will then be used to route back to the node the contextual information that can be collected from the other nodes in the network. Such an approach may be very powerful for locally gathering contextual information without strict locality constraint and without requiring any centralized service (all nodes cooperate to provide the information in a distributed way). Although very interesting, we think that this proposal lacks flexibility, in that it is purely focused on information acquisition, and pays little or no attention to the problem of coordinating the activities of application components. In our opinion, TOTA is more flexible: the possibility of programming propagation rules makes it possible to express complex coordination patterns other than information gathering only.

The ObjectPlaces middleware [43] shares very similar goals to that of EgoSpaces, 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 access via object-matching. However, and this is where the relation with EgoSpaces and TOTA comes in, ObjectPlaces enables associating to any object stored in a local tuple space a sort of distributed data structure, called "view". Such distributed data structure propagates around the object in the network and define a sort of field enabling to detect where an object is, i.e., in which direction and at what distance in the network. ObjectPlaces, while being quite flexible in the definition of how to build and propagate such distributed data structures, lacks the power and flexibility of the TOTA programming approach.

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 [35, 13]. 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 towards 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 [45], as well as to orchestrate the activities in groups of mobile robots [44]. Although being focused on very specific application scenarios, all these approaches share with TOTA the idea of using distributed data structures to drive agents actions.

7 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 paper, 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 coordination patterns can be easily enforced in a self-organizing and self-adaptive way; *(iii)* the TOTA middleware, while being light-weight, effectively supports network dynamics by automatically reshaping tuples' distribution accordingly 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 accesses to distributed tuples and their updates, and to protect privacy of data and of actions. Indeed, security and privacy are challenging issue for the whole area of pervasive and mobile computing, and we are confident several emerging proposals in the area of mobile ad-hoc networks [33] and pervasive computing [42] 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 [34]), 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. 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 distributed computing [11, 48], in which the exploited coordination models are mostly based on the diffusion of virtual chemical gradients or virtual force fields (and thus directly corresponding to the model enforced by TOTA), will help paving the way for the identification of such methodologies.

References

[1] Familiar linux. http://familiar.handhelds.org.

[2] Gnutella. http://gnutella.wego.com.

[3] Green light district. http://stoplicht.sourceforge.net.

[4] International workshop on mobile teamwork support. http://www.infosys.tuwien.ac.at/motion/mts.

[5] Java 2 micro edition (j2me). http://www.java.sun.com.

[6] Java bot for unreal tournament. http://utbot.sourceforge.net.

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

[8] Jprofiler. http://www.ej-technologies.com/products/jprofiler/overview.html.

[9] Modular reconfigurable robotics at parc. http://www2.parc.com/spl/projects/modrobots.

[10] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[11] O. Babaoglu, G. Canright, A. Deutsch, G. Di Caro, F. Ducatelle, L. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, and A. Montresor. Design patterns from biology for distributed computing. In *European Conference on Complex Systems*. Paris, France, 2005.

[12] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *International Conference on Distributed Computing Systems*. IEEE CS Press, Columbus (OH), USA, 2005.

[13] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[14] F. Bellifemine, A. Poggi, and G. Rimassa. Jade - a fipa2000 compliant agent development environment. In *Proceedings of the International Conference on Autonomous Agents (Agents 2001)*. ACM Press, Montreal, Canada, 2001.

[15] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence. From Natural to Artificial Systems*. Oxford University Press, Oxford, United Kingdom, 1999.

[16] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode. Cooperative computing for distributed embedded systems. In *22nd International Conference on Distributed Computing Systems*. IEEE CS Press, Vienna, Austria, 2002.

[17] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *24th International Conference on Distributed Computing Systems*. IEEE CS Press, Tokio, Japan, 2004.

[18] G. Cabri, L. Leonardi, M. Mamei, and F. Zambonelli. Location-dependent services for mobile users. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems And Humans*, 33(6):667 – 681, 2003.

[19] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332 – 383, 2001.

[20] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco. Mobile data collection in sensor networks: The tinylime middleware. *Journal of Pervasive and Mobile Computing*, 4(1):446–469, 2005.

[21] N. Davies, K. Cheverst, K. Mitchell, and A. Efrat. Using and determining location in a context-sensitive tour guide. *IEEE Computer*, 34(8):35 – 41, 2001.

[22] D. Estrin, D. Culler, K. Pister, and G. Sukjatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59 – 69, 2002.

[23] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114 – 131, 2003.

[24] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. Complex behavior at scale: An experimental study of lowpower wireless sensor networks. In *Technical Report UCLA/CSD-TR 02-0013*. Los Angeles (CA), USA, 2002.

[25] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57 – 66, 2001.

[26] B. Johanson and A. Fox. The event heap: A coordination infrastructure for interactive workspaces. In *4th IEEE Workshop on Mobile Computer Systems and Applications (WMCSA)*. IEEE CS Press, Callicoon (NY), USA, 2002.

[27] C. Julien, C. Roman, and J. Payton. Context-sensitive access control for open mobile agent systems. In *3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS)*. Edinburgh, United Kingdom, 2004.

[28] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[29] M. Mamei and F. Zambonelli. Self-maintained overlay data structures for pervasive autonomic services. In *2nd IEEE Workshop on Self-managed Networks, Systems, and Services*. Springer Verlag, Dublin, Ireland, 2006.

[30] M. Mamei, F. Zambonelli, and L. Leonardi. Distributed motion coordination with co-fields: A case study in urban traffic management. In *6th IEEE Symposium on Autonomous Decentralized Systems*. IEEE CS Press, Pisa, Italy, 2003.

[31] M. Mamei, F. Zambonelli, and L. Leonardi. Co-fields: A physically inspired approach to distributed motion coordination. *IEEE Pervasive Computing*, 3(2):52 – 61, 2004.

[32] C. Mascolo, L. Capra, Z. Zachariadis, and W. Emmerich. Xmiddle: A data-sharing middleware for mobile computing. *Wireless Personal Communications*, 21:77 – 103, 2002.

[33] A. Mishra and K. M. Nadkarni. Security in wireless ad hoc networks. pages 499–549, 2003.

[34] A. Montresor, M. Jelasity, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[35] R. Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Proceedings of the 1st Joint Conference on Autonomous Agents and Multi-Agent Systems*. ACM Press, Bologna, Italy, 2002.

[36] R. Nagpal, H. Shrobe, and J. Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *2nd Intl Workshop on Information Processing in Sensor Networks*. 2003.

[37] R. O'Grady, R. Groß, F. Mondada, M. Bonani, and M. Dorigo. Self-assembly on demand in a group of physical autonomous mobile robots navigating rough terrain. In *8th European Conference on Artificial Life*, 2005.

[38] H. V. Parunak. Go to the ant: Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75:69 – 101, 1997.

[39] G. Picco, A. Murphy, and G. Roman. Lime: a coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15, 2006.

[40] K. Pister. On the limits and applicability of mems technology. In *Defense Science Study Group Report, Institute for Defense Analysis*. Alexandria (VA), USA, 2000.

[41] S. Ratsanamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *ACM SIGCOMM Conference*. ACM Press, San Diego (CA), USA, 2001.

[42] P. Robinson, H. Vogt, and W. Wagealla. *Privacy, Security and Trust within the Context of Pervasive Computing*. Springer Verlag, Berlin, Germany, 2005.

[43] K. Schelfthout, T. Holvoet, and Y. Berbers. Views: Customizable abstractions for context-aware applications in manets. In *Proceedings of the 3rd International ICSE*

Workshop on Software Engineering for Large Multiagent Systems. Springer Verlag, St. Louis (WS), USA, 2005.

[44] W.-M. Shen, P. M. Will, A. Galstyan, and C.-M. Chuong. Hormone-inspired self-organization and distributed control of robotic swarms. *Autonomous Robots*, 17(1):93–105, 2004.

[45] K. Stoy and R. Nagpal. Self-reconfiguration using directed growth. In *7th International Symposium on Distributed Autonomous Robotic Systems*. Toulouse, France, 2004.

[46] R. Want and G. Borriello. Survey on information appliances. *IEEE Computer Graphics and Applications*, 20(3):24 – 31, 2000.

[47] D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of egv transportation systems. In *5th International Conference on Autonomous Agents and Multiagent Systems – Industry Track*. ACM Press, Utrecht, Netherland, 2005.

[48] F. Zambonelli, M. Gleizes, M. Mamei, and R. Tolksdorf. Spray computers: Explorations in self-organization. *Journal of Pervasive and Mobile Computing*, 1(1):1 – 20, 2005.