A Programmable Event-based Middleware for Pervasive Mobile Agent Organizations

Matteo Gazzotti, Marco Mamei, Franco Zambonelli

Dipartimento di Scienze e Metodi dell'Ingegneria – Università di Modena e Reggio Emilia

Via Allegri 13 – 42100 Reggio Emilia – ITALY

matteo_gazzotti@libero.it

{ mamei.marco, franco.zambonelli}@unimo.it

Abstract

This paper firstly introduces a conceptual framework for the effective design and development of distributed pervasive applications based on mobile agents. The framework, based on the definition of active organizational contexts, promotes an engineered and modular approach to application design by introducing the notion of active organizational contexts. Then, the paper describes the architecture and the implementation of a re-configurable event-based micro-kernel implementing active organizational context, suitable as a supporting middleware for pervasive applications based on mobile agents. An application example in the area of urban traffic control shows the effectiveness of the approach.

1. Introduction

Computing is becoming pervasive. Autonomous computer-based systems are going to be embedded in all our everyday objects and in our physical environment, and they are going to interact with each other in a globally connected network, possibly making use of wireless communication technologies [4, 15]. In such a scenario, mobility too, in different forms, will be pervasive [1]. Mobile users, mobile devices, transportable computer-based objects, as well as mobile software components, define an open and dynamic networked world, in which the topology of interactions change with time. As a consequence, the effects of computation and coordination activities are likely to dramatically change depending on the location, i.e., the context, in which they occur.

Defining suitable models and infrastructures for the design and development of distributed applications in such pervasive and mobile scenario is indeed an open research challenge. Nevertheless, it is being widely recognized that, in order to limit complexity of application design and development it is necessary to define models and infrastructures enabling to *(i)* handle mobility in a natural and uniform way and *(ii)* support effective and flexible coordination despite the dynamics of interaction networks.

With regard to the former issue, a promising approach is to model application components, as well as physical mobile devices and computer-based systems, in terms of *autonomous mobile agents* [18]. In fact, the observable behavior of a software process running on a computer-based mobile device is that of an autonomous mobile component having local control over its activities. With regard to the latter issue, we propose extending the notion of "sociality" intrinsic in agent-based computing by explicitly modeling agents in terms of "organizational" entities. In particular, we consider mobility as a movement across organizations, more than simply across locations: the coordination activities of mobile agents change not only due to the different entities that they find in different locations, but also due to the different coordination laws to which they must obey in different organizations. Such a conceptualization considers the context in which an agent executes as an active context, capable of influencing agents' coordination activities. As described in Section 2, this can promote a clean separation of concerns between computation and coordination [6], simplifying and making more modular application design.

Starting from the above conceptual organizational framework, the paper shows how it can be supported, in application development, by a programmable event-based middleware infrastructures, in which all interactions can be expressed in terms of event generations and subscriptions, and in which the effect of an event can be properly programmed to enact specific coordination laws and influence local coordination activities. The general concepts underlying such event-based infrastructure are described in Section 3. A light micro-kernel based

implementation of the infrastructure, easy to use and also suitable for resource limited devices (as those that can be found in pervasive scenarios) is presented in Section 4. An application example in the area of traffic management is described in Section 5 to clarify the concepts expressed and to show the effectiveness of our approach.

Eventually, Section 6 discusses related work and Section 7 concludes the paper.

2. The Conceptual Framework

2.1. Local Interaction Context

As far as the high-level issues related to the modeling, design and development of complex pervasive applications are concerned, handling mobility is basically a problem of managing the coordination activities of application agents, i.e., all the activities of agents that may influence the surrounding computational world or, which is the same, all of the events during the execution of agents that can be perceived from the external. These events may include the arrival/departure of an agent from a given location (computational, as a Web site, or physical, as a building), the accesses by agents to the local resources, and the communication and synchronization activities with the other agents of the location, whether of the same application or foreign agents.

In general mobility can be modeled in terms of movements across *local interaction contexts*. A local interaction context defines the agents' perceivable world, which changes depending on the agent position, and which represents the logical place in which all agents' coordination activities occur. What is the interaction model to be actually exploited by agents for modeling interaction in a context is not of primary influence for the sakes of application modeling. Interactions may occur via message passing and ACLs [5], via meetings [16], or via shared dataspaces [1]. The two key points that really matters from the software engineering perspective are that:

- the enforced-locality-model reflects, at the level of application modeling, a notion of context intrinsic in mobility and embedded-pervasive computing.
- the notion of local interaction context provides a useful conceptual abstraction for analyzing the interactions between a set of agents in a local site in terms of the observable behavior of the agents activities, i.e., in terms of all the interaction events occurring locally to a site.

2.2. Local Organizations

Movements across local interaction contexts may impact on agents' coordination activities. In fact, coordination activities are likely to be strictly ruled by proper security and resource control policies, which may be different from location to location, i.e., from a local interaction context to another. In such a scenario, the local interaction context cannot be simply considered as the place in which coordination activities occur. Instead, a local interaction context has to be considered an *active context* (or active environment), capable of enacting specific *local coordination laws* to rule and support the agents' coordination activities.

By assuming an organizational perspective [17], one can consider the local interaction context in terms of an organizational context: an agent, by entering via a movement in an interaction context, enters a foreign organization where specific organizational rules are likely to be enacted in the form of coordination laws. Thus, for the sake of conceptual simplicity, one must consider the interaction context as the locus in which the organizational laws ruling the activities of the local organization reside [17]. For example, when new kinds of application agents are going to be deployed on a local interaction context, the administrator of that organizational context can analyze which local coordination laws that (s)he may find it necessary to locally enforce. These can be used both to facilitate the execution of the agents on a site and to protect it from improper exploitation of the local interaction context.

2.3. Application-Specific Organizations

The above is not the full picture. In fact, agents may be part of a cooperative multi-agent application, and move in a pervasive network to cooperatively achieve, according to specific protocols and patterns, specific application goals. Given that, it is clear that agents' coordination activities within a multi-agent application may be required to occur accordingly to specific laws ruling the whole application and ensuring the proper achievement of the application goals. In other words agents belonging to a specific application logically constitute an application-specific organizational context on their own.

Within the organizational perspective, this implies that the context in which an agent executes and interacts is not only the one identified by the local organizational context, but is also the one of its own multiagent organization. As that, a local organizational context should not only be thought as the place in which local organizational rules reside, but also as the active context in which application agents may enact their own,

application-specific, organizational rules in the form of coordination laws. These application-specific coordination laws, are not located in a single local interaction context, but are in principle spread (i.e. replicated) through all local interaction contexts interest to the application, to be enforced independently form agents' current location.

2.4. Impact on Application Design

The above analysis suggests modeling and designing applications in terms of agents interacting via *active organizational contexts* (see Figure 1).

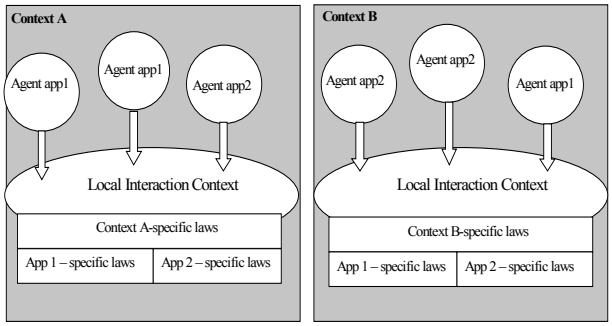

Fig. 1. **Local Organizational Contexts.**

The adoption of such a conceptual framework – that we call *context-dependent coordination* [1] – can have a very positive impact on the engineering of mobile agent applications. From the point of view of application designers, the framework naturally invites in designing an application by clearly separating the intra-agent aspects and inter-agent (organizational) ones. The formers define the internal behavior of agents and their observable behavior. The latters define the application-specific organizational laws according to which agents should interact with each other and with external entities for the global application goal to be coherently achieved, and lead to the identification of the coordination laws that agents should spread on the visited organizational contexts. This separation of concerns is likely to reduce the complexity of application design, and can make it more modular and easy to be maintained (design-for-change perspective).

3. Towards a Programmable Event-based Middleware

The separation of concerns promoted during design can be preserved during the development and maintenance phases too if a proper coordination middleware infrastructure is available that somehow reflects the concepts and the abstractions of context-dependent coordination. In that case, the code of the agents can be clearly separated from the code implementing the coordination laws (whether local organizational laws or application-specific ones). Thus, agents and coordination laws can be coded, changed, and re-used, independently of each other.

3.1. Programmable Coordination Media

A coordination infrastructure for context-dependent coordination must be based on an architecture relying on a multiplicity of independent interaction abstractions, each implementing the notion of active organizational contexts. Such interaction abstractions can be effectively implemented in terms of *programmable coordination media* [3]: a software in charge of monitoring, mediating and ruling all coordination activities of application agents within a locality, accordingly to local (or application-specific) coordination laws, embedded into the medium itself. Whatever the interaction model it relies upon, a coordination medium is generally characterized by:

(i) a set of primitive operations to let agents access it;
(ii) an internal behavior, intended as the computational activity performed inside the coordination medium in response to interactions events, i.e., in response to the invocation of a specific primitive performed by an agent.

Most of the existing coordination infrastructures fix the internal behavior of coordination media once and for all: the behavior of a coordination medium in response to a given interaction event is always the same. An infrastructure based on *programmable coordination media* [3, 10] makes it possible – without changing the set of primitive operations used by agents to access the coordination media – to program the internal behavior of a coordination medium and override its default behavior so as to adapt it to the specific needs of applications or of the local environment. To this end, one must:

(i) fully characterize the kind of access event of interest, in terms of the identity of the agent performing it, the primitive used to access the coordination medium, and the parameters possibly supplied in the invocation of the primitive;
(ii) express the new behavior (we usually called it "reaction") to be assumed by the coordination medium in response to this kind of access event, and have this behavior override the default one.

3.2. Programmable Event-based Media

From the above characterization of programmable coordination media, it should be clear that it does not really matter what are the specific primitives used by agents to access the media. From a conceptual point of

view, a programmable coordination media can be realized upon any given interaction model. However, neither message-passing not shared dataspace models appears suitable for mobile and pervasive scenarios: the former lacks the necessary uncoupling required to deal with the dynamics of the scenario, the latter may require too large memories for resource limited devices, as those that may be found in pervasive scenarios. Event-based models, being able to put entities in indirect contact without requiring large memory spaces, appears a good trade-off for adoption in pervasive and mobile scenarios, as testified by several recent proposals in the field [2, 14]. Moreover, as formally described in [20] the publish-subscribe event-based interaction model, despite its simplicity, has the full expressive power of more complex interaction models. Thus, it is always possible to exploit an event-based coordination media to build an additional layer above it, to let agents interact accordingly to, say, a shared data space or a message-passing interaction model.

Therefore, we propose having agents interact in a context by generating events and by subscribing to events. A local event-based programmable coordination media is in charge of receiving events, re-distributing them, and performing computations accordingly to the specific behavior programmed in it (expressing either application-specific or local organizational laws).

The first basic functionality provided by the coordination media is to act as an event dispatcher for the interaction context. Following the traditional publish/subscribe event model [2], we characterize an event by a set of parameters, using the following notation: $E(par_1,\ldots,par_n)$. Events can be generated by agents either explicitly (by specific instructions in the executing code) or implicitly (due to the occurrence of specific condition, as detailed in the following). Subscriptions are issued by agents to express their interest in particular (class of) events and we indicate them with the notation $\hat{E}(E(par_1,\ldots,par_n),null)$: to specify that the agent is interested to events matching the event signature within the subscription. Once the agent is notified by the event-kernel about the occurrence of an event it can react accordingly, as prescribed in its internal code. Moreover, agents and site administrators can implant coordination laws in the coordination media. Exploiting the semantic of the event-model, this can be done by performing the following kind of subscription: $\hat{E}(E(par_1,\ldots,par_n),Rct)$: that means that the reaction *Rct* (which can be an arbitrary piece of code) will be executed by the middleware when the event $E(par_1,\ldots,par_n)$ is fired. The middleware acts simply as another component capable of reacting to the events produced. From a conceptual point of view, the middleware can be thought as a passive component whose subscriptions and reactions are decided by others (agents or administrators).

To provide flexibility to the event model, we finally specified the use of wild-cards (*star* values) within events and subscriptions descriptions. For example, the subscription $\hat{E}(E(*,y,*),null)$ specifies an interest to all the events described by three parameters, having *y* as second parameter. Events like $E(*,X)$, and subscriptions like $\hat{E}(E(A,*),null)$ match, because the *star* value in the event is matched with the *A* value of the subscription, while the *star* value of the subscription is matched with the *X* value of the event.

The event-based media can serve not only as a place to catch interaction events occurred due to the explicit invocation of an event-generation primitive, but also to deal with implicitly generated events. In fact, agents can generate events (i.e., can be made observable from the external) also due to specific lifecycle changes. In the presence of mobility, the arrival and the departure of agents to/from interaction contexts typically generate events that may be in need to be managed. Also in these cases, these events can be characterized by different parameters (e.g., specifying the identity and the nature of the agent, its budget, etc.), and it is possible to program the event-kernel so as to react to them with specific actions. However, it is worth emphasizing that modeling the arrival/departure of an agent in an interaction context in terms of events once again enforces a uniform and general way of handling mobility. In fact, from the point of view of the interaction context and of the events it perceives, there is no difference between, e.g., the arrival of a mobile wireless device in the transmissions range of the kernel access point and the arrival of a mobile Java thread in the computer hosting the kernel. If necessary, specific event parameters can be used to associate different event handlers for different types of mobility.

4. Implementation of a Programmable Event-based Micro Kernel

Accordingly to the event-based model described in the previous section, we have implemented an event-based programmable coordination media in terms of a programmable micro event-based kernel. The kernel acts as an engine to process the events occurring in a local interaction context accordingly with the programmed behavior, as resulting from the inserted subscriptions. The implementation is fully in Java, executable with Personal Java [13], and currently supporting IEEE 802.11b wireless connections.

4.1. The Architecture

The duty of the kernel is to process events and, via pattern matching, to check subscriptions and execute the associated reactions. To obtain this behavior, the kernel implements a simple and intuitive architecture entirely developed in Java (Figure 2). The dramatic simplicity of the architecture, and the consequent very light load (in terms of both storage space and computational activity) imposed on the hosting computers, makes it very suitable for pervasive computing scenarios. The event kernel can be installed in almost every type of embedded or wireless mobile computing device supporting Personal Java.

The events caught in the local context are stored in an input FCFS *event queue*. A thread called *puller thread* constantly examines the state of the queue and, if there is at least one event, it passes the event to *kernel engine*. The kernel engine kernel implements a pattern-matching operation for all processed event, to check for subscription matching the event. All active subscriptions are registered in the *subscription list*. It is worth to remember that due to *star*-values that event parameters can assume, a reaction inserted in a subscription could be triggered by a classes of events as well as specific events.

When a matching is verified, the associated reaction (or reactions) has to be performed. The kernel engine is devoted to spawn a thread perform the code of the reaction. The code of a reaction is not limited to operate outside the kernel, but it is also enabled to operate on the internal subscription list, inserting or removing subscriptions. Furthermore the code of a reaction can generate other events that are inserted in the event queue.

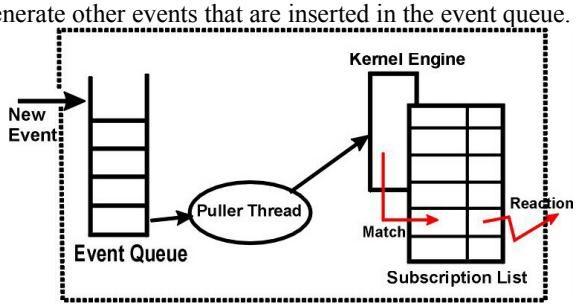

Fig. 2. **The architecture of the Micro-kernel**

4.2. Implementation of events and reactions

The structure of the events manageable of a kernel is a key concept: it is important to realize a light and flexible kernel, which must be able to accept any kinds of application-specific events.

Accordingly to the OO paradigm, all events are objects instances of some classes; the Java package that contains the core kernel classes defines only a simple hierarchy of events classes. Any application in need to use the kernel can extend event classes to obtain specific and suitable event types, inserting new parameter and/or methods. Because all the structures of kernel (queue, engine, subscription list) are designed to work with references to superclasses of these application-event, there are no problems to manage these references; moreover the polymorphism of Java permits at the specific methods of the event subclasses to be called directly and automatically from super-classes references.

Also the concept of reaction has been well defined: a reaction can be an arbitrary piece of code, enabled to interact with kernel programmed behavior. The kernel defines a generic reaction in terms of an abstract class declaring an abstract method with a known name (*performReaction*); the subclasses of this class implements this method, realizing a specific and arbitrary type of reaction. Such a constrained interface allows the kernel to perform the code of any reaction, by invoking the known method when the reaction is triggered.

4.3. Using the Event-based Kernel

To use the event-based kernel, a dramatically simple yet effective interface is provided. Only two basic operations are provided to generate an event to be caught by the kernel and to subscribe to an event (thus possibly programming the event kernel), respectively.

To generate an event, one has to generate an object of an event class, e.g.,

FooEvent fe = new FooEvent(par1, …, parn);

and then send this event to the kernel via the notifyEvent primitive:

kernel.notifyEvent(fe);

Where it is assumed that the *Kernel* reference always points to the kernel of the current interaction context (this is simply achieved via a Jini-like discovery process [7]). As already discussed, other types of events may be generated in an implicit way and automatically notified to the kernel.

To subscribe to an event one has to generate a reaction object, e.g.,

FooReaction fr = new CrashReaction();

a template event, with possibly some non defined parameters, e.g.,

FooEvent fe = new FooEvent(par1,, *, …, parn);*

and eventually invoke the *addSubscription* primitive:

kernel.addSubscription(fe, fr);

Where the second parameter could also be *null*, to have a simple expression of interest rather than a programming of the event kernel.

4.4. Providing a State to the Micro Kernel

The only persistent stateful component of the kernel is the subscription list containing only the couples

(*event→reaction*). Such architecture implies a generally stateless behavior of the coordination laws programmed in the form of reactions. This kind of behavior is enough, for example, for event dispatching among agents living in the same interaction context. However, for others types of applications there may be the need of a persistent state for historical events or for data.

In our implementation, exploiting the versatility of the reaction mechanism, which can access any object outside the kernel, can easily satisfy this need. Thus, a data/state space can be implemented outside the kernel in terms of an object, and made it accessible from a generic reaction (e.g., by providing the reaction object with the reference to the state object).

From a different viewpoint, the data state can also be considered a non-mobile agent, near the kernel; this point of view has at least two advantages: *(i)* the kernel can treat all external entities in a uniform way, as agents.; *(ii)* the state can be itself "active": is possible for it to send events to kernel or insert/remove subscriptions.

5. Application Example: Traffic Management

5.1. Scenario

The growth of motor traffic, especially in metropolitan areas, calls for effective ways to control and support to traffic circulation, and aimed both at obtaining a rational use of roads and infrastructures (e.g., traffic lights, tunnel, roundabout, etc..) and at giving drivers prompt, useful and personalized information. GPS gives users the capabilities to dynamically obtain traffic information. However, to realize an optimized use of traffic infrastructures, GPS isn't enough. For example, a traffic light must know the real-time situation of vehicles near the cross to calculate dynamically the optimal temporization time that ensures the maximum flow of traffic trough the cross. Also, there is need for bi-directional exchange of information between vehicles and traffic infrastructures, to provide dynamic and informed coordination among vehicles.

In this scenario, when developing an application for traffic control, wireless-enabled computers on vehicles play the role of mobile agents. From the infrastructure point of view, we can assume the presence of computer- and wireless- enriched traffic-lights, roundabout, etc., and we can think to allocating a kernel in these computer nodes, to act as the organizational context in which the agents/vehicles in their proximities (i.e., in their connection range) execute, and via which they coordinate with each other (Figure 3-a).

In extra-urban context, the lack of infrastructure elements of reference where to allocate kernels may require a different solution. Agents have to interact with each other directly, in an ad-hoc network: each agent must host an own version of the kernel, to filter and react to events coming from the other vehicles/agents in the connection range, and to distributed them the event it generates (Figure 3-b).

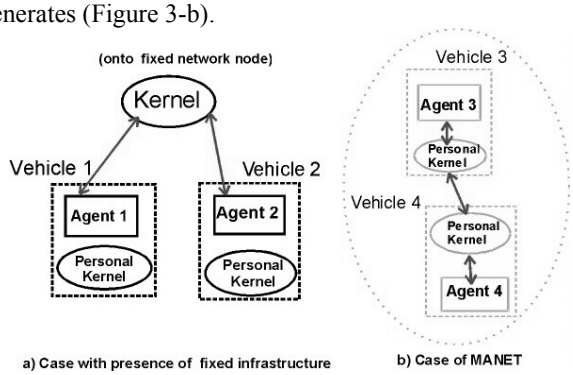

Fig. 3. **Kernel-based Coordination for Traffic Control**

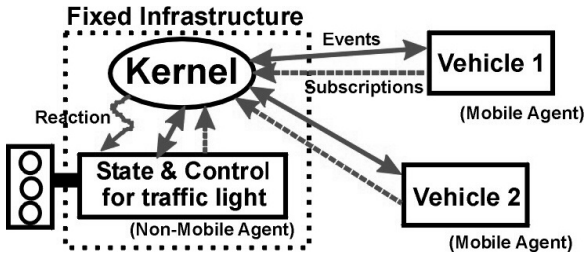

Fig. 4. **Intelligent Traffic Lights**

5.2. Programming Intelligent Traffic Lights

Putting attention to the urban case, where infrastructure reference elements are available, we have developed a prototypal model of an intelligent traffic light (Figure 4), and simulated its behavior in the presence of different traffic conditions. In this application is necessary for a kernel to have a persistent state to perform parametric reactions, for example it must be able to discriminate among the states of traffic light: how we previously discussed, a persistent state is outside a kernel, but it can be accessed from his reactions.

The programmable event-kernel enhances the functionality of a simple traffic light: for instance it can be programmed to dynamically calculate a temporization in function the traffic condition. Moreover, it enables vehicles to subscribe to specific traffic conditions and to enable a vehicle to signal its presence to other vehicles in the proximities. As a very simple example, when a vehicle crash occurs, it could be useful to alert other vehicles in the proximities. This goal can be reached defining a simple reaction class, *CrashReaction*, whose

code is shown in figure 5.

Let us suppose that, when a vehicle crashes, it generates an event of the type *VehicleEvent* with a specific "*Crash*" string in its second field, and that is event is sent to the traffic light in reach (or, in the case of a Mobile Ad-hoc NETwork – MANET, to all reachable agents). Such an event generation and notification could be implemented by the following code:

```
VehicleEvent event = new VehicleEvent (myId, "Crash",
vehicleType,currentSpeed,position);
kernel.notifyEvent(event);
```

In order to trigger this code when an event crash occurs, an object of the *CrashReaction* has to be created and a proper subscription of such a reaction can added to the event kernel via, e.g., the following code:

```
CrashReaction crashReaction = new CrashReaction();
kernel.addSubscription(new
VehicleEvent(*,"Crash",*,*,*,*), crashReaction);
```

which specify to that the execution code of the *crashReaction* code has to be associated whenever a *VehicEvent* occurs that, in its second field, matched the string "*Crash*". Such a subscription can be added to a traffic light by any authorized agent, i.e., from a policeman or by the urban traffic administrators, thus assuming the form of local organizational laws of the traffic light. However, it is also possible to think at having similar application-specific laws, specific for, say, a local taxi company, which prefer to handle in a different way (i.e., with additional reactions) a crash event and the notification for its taxies.

```
public class CrashReaction extends Reaction{
 public CrashReaction()
 {super();}

 public void performReaction(BasicEvent e, Kernel k) {
 //gain all agent references to perform event broadcast
  ArrayList al = k.getAllAgentInterfaces();
 for (int i=0;i<al.size();i++)
   try
    {//send event to all agents in local context
     ((RemoteEventListener)al.get(i)).notify(e);
    }catch (Exception exc)
       {System.out.println("ERROR: agent not found");}
 }
}
```

Fig. 5**. The CrashReaction code.**

Of course, the event kernel can also be enriched by a simple graphical interface to add subscription and to monitor current and past events. Figure 6 shows the special-purpose graphical interface provided with the intelligent traffic light application, with which to monitor and program the traffic light kernel. Such interface has been defined as a simple extension of a general-purpose interface provided with the kernel.

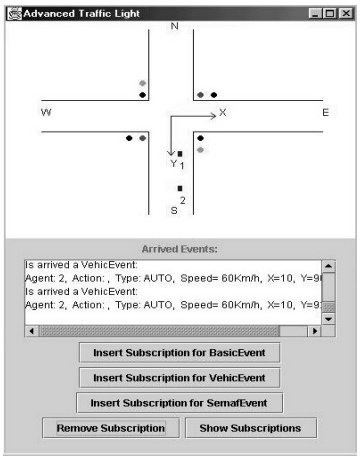

Fig. 6**. Traffic Control Graphical Interface**

6. Related Work

Coordination middleware based on a variety of programmable coordination media can be found in the literature. Tucson [11] and MARS [1], developed in the context of an affiliated project, exploits programmable tuple spaces as coordination media. However, they lack the identification of the organizational approach, and, being based on rather heavy architectures, are not suitable for pervasive computing scenarios. The LGI model [10] proposes influencing the behavior of agents by dynamically attaching coordination laws to agents interacting via message-passing, thus defining some sorts of message-based coordination media. However, LGI does not take into account mobility at all. LIME [12] defines an interesting and peculiar tuple-based architecture for handling in a uniform way both physical and actual agent mobility, also in the context of MANETs. However, LIME integrates only very limited forms of reactivity: in fact, LIME does not reach the full programmability required for context-dependent coordination and does not promote in any case an organizational perspective. A more recent proposals for MANETs scenario, XMIDDLE [9], rely on shared accesses to tree-like distributed data structures where a limited form of programmability is defined only to enforce consistency in disconnected operations over the data structure.

Coming to event-based infrastructures, most of the recent proposals in the area focus on large-scale distributed applications with the main aim of providing suitable distributed architecture for wide-scale event and subscription dispatching. A notable example is the JEDI system, described in [2], and where an extensive discussion of other event-based architectures can be

found. However, the few proposals specifically focusing on mobility aims only at enabling event dispatching in the presence of mobility, rather than at defining programming models enforcing locality and suitable for mobility. In addition, to our knowledge, none of them identify the need for programmability. In our approach, we can exploit the programmability of the kernel to implement, as a high-level service, any algorithm for wide-area dispatching of events and subscriptions. A different category of event-based middleware proposals focus on content-based networking [14], with the aim of providing a model of dispatching of event and subscription relying on the content of events rather than on the identities of the involved processes. Again, in our approach, programmability can be exploited to implement any needed context-based dispatching policy.

7. Conclusions and Future Work

The paper has introduced a conceptual organization-oriented framework for the design of mobile agents application in pervasive computing scenarios. On this basis, it has described the architecture of an event-based programmable architecture, based on the definition of a minimal event-kernel, thus suitable for deployment also in resource-constrained devices.

Our current research work has two objectives. On the one hand, we are studying high-level coordination model, to be provided in terms of special-purpose services programmed into the event-based kernel, for the global orchestration of the movements in a workgroup (e.g., in a rescue team) [8]. On the other hand, we are investigating the effectiveness of the proposed model in ruling and controlling coordination activities in systems with a massive number of agents/components [19].

Acknowledgments. Work partially supported by Nokia Research Center Boston and by MIUR Project "MUSIQUE".

References

[1] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", IEEE Transactions on Software Engineering, 28(9), September 2002.

[2] G. Cugola, A. Fuggetta, E. De Nitto, "The JEDI Event-based Infrastructure", IEEE Transactions on Software Engineering, 27(8), August 2001.

[3] E. Denti, A. Natali, A. Omicini, "On the Expressive Power of a Language for Programmable Coordination Media", Proceedings of the 10th ACM Symposium on Applied Computing, ACM, 1998.

[4] D. Estrin, D. Culler, K. Pister, G. Sukjatme, "Connecting the Physical World with Pervasive Networks", IEEE Pervasive Computing, 1(1), Jan. 2002.

[5] T. Finin at al., "KQML as an Agent Communication Language", 3rd International Conference on Information Knowledge and Management", November 1994.

[6] D. Gelernter, N.Carriero "Coordination Languages and Their Significance", Communication of the ACM, Vol. 35, No. 2, pp. 96-107, February 1992.

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

[8] M. Mamei, L. Leonardi, F. Zambonelli, "A Physically Grounded Approach to Coordinate Movements in a Team", 1st International Workshop on Mobile Teamwork, Vienna (A), July 2002.

[9] C. Mascolo, L. Capra, W. Emmerich, "An XML based Middleware for Peer-to-Peer Computing", In Proc. of the International Conference of Peer-to-Peer Computing, IEEE CS Press.

[10] N.H. Minsky, V. Ungureanu, "Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems", ACM Transactions on Software Engineering and Methodology, 9(3), 2000.

[11] A. Omicini, F. Zambonelli, "Coordination for Internet Application Development", Autonomous Agents and Multiagent Systems, 2(3), Sept. 1999.

[12] G.P. Picco, A.M. Murphy, G. -C. Roman, "LIME: A Middleware for Logical and Physical Mobility", 13th International Conference on Distributed Computing Systems, Linda Meets Mobility", July 2001.

[13] http://java.sun.com/products/personaljava

[14] A. Rowstron, P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems", 18th ACM Conference on Distributed Systems Platforms, Heidelberg (D), Nov. 2001.

[15] D. Tennenhouse, "Proactive Computing", Communications of the ACM, 43(5), May 2000.

[16] J. White, "Mobile Agents", in Software Agents, J. Bradshaw (Ed.), AAAI Press, pp. 437-472, 1997.

[17] F. Zambonelli, N. R. Jennings, M. J. Wooldridge, "Organizational Abstractions for the Analysis and Design of Multi-agent Systems", in Agent-Oriented Software Engineering, LNCS No. 1947, 2001.

[18] F. Zambonelli, V. Parunak, "From Design to Intentions: Sign of a Revolution", 1st International Joint Conference on Autonomous Agents and Multi-agent Systems, Bologna (I), July 2002.

[19] F. Zambonelli, A. Roli, M. Mamei, "Dissipative Cellular Automata As Minimalist Distributed Systems: A Study On Emergent Behaviors", 11th Euromicro Conference on Parallel Distributed and Network based Processing, Genoa (I), Feb 2003.

[20] G. Zavattaro, N. Busi, "Publish/Subscribe vs. Shared Dataspace Coordination Infrastructures", 10th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, Boston (MA), June 2001.