Location-Dependent Services for Mobile Users

Giacomo Cabri, Letizia Leonardi, Marco Mamei, and Franco Zambonelli, *Member, IEEE*

***Abstract*—One of the main issues in mobile services' research (M-service) is supporting M-service availability, regardless of the user's context (physical location, device employed, etc.). However, most scenarios also require the enforcement of context-awareness, to dynamically adapt M-services depending on the context in which they are requested. In this paper, we focus on the problem of adapting M-services depending on the users' location, whether physical (in space) or logical (within a specific distributed group/application). To this end, we propose a framework to model users' location via a multiplicity of local and active service contexts. First, service contexts represent the mean to access to M-services available within a physical locality. This leads to an intrinsic dependency of M-service on the users' physical location. Second, the execution of service contexts can be tuned depending on who is requesting what M-service. This enables adapting M-services to the logical location of users (e.g., a request can lead to different executions for users belonging to different groups/applications). The paper firstly describes the framework in general terms, showing how it can facilitate the design of distributed applications involving mobile users as well as mobile agents. Then, it shows how the MARS coordination middleware, implementing service contexts in terms of programmable tuple spaces, can be used to develop and deploy applications and M-services coherently with the above framework. A case study is introduced and discussed through the paper to clarify our approach and to show its effectiveness.**

***Index Terms*—Context-awareness, coordination infrastructures, M-services, mobility, multiagent systems.**

I. Introduction

MOBILITY is becoming a *sine qua non* characteristic of current information and communication technologies scenarios, and several efforts are being spent to support mobility in different areas. In particular, mobile service (M-service) technologies focus on the increasing need to support the ubiquitous provisioning of electronic services to mobile users, i.e., to nomadic users carrying some sorts of portable computer-based devices.

In the past few years, the mainstream focus of M-service researches has been rooted on the *anytime-anywhere* principle: requests for services by mobile users should be always satisfied in an unchanged and transparent way, regardless of the time at which a service is requested and of the place from which it is requested [22], [23]. However, this is not the full picture. In fact, in several cases, the design and development of effective M-services should explicitly take into account the characteristics of the *context* from which a service is requested [15]. These characteristics may include, among the others: the type of device exploited to access a service, the physical location of the user, its personal preferences, or the preferences of a group to which the user belongs. In general, the capability of designing and deploying *context-aware services* [20], [30], i.e., services whose execution can be dynamically adapted to the characteristics of the context, can provide a strongly added value to M-services technologies. For instance, a tourist visiting a town may find extremely useful to have access to services that automatically adapt the type and format of information provided depending on the capabilities of his/her display (e.g., automatically selecting appropriate fonts), on his/her current location (e.g., providing information only about reasonably close restaurants), and on his/her current personal budget (e.g., providing information about cheap restaurants).

In this paper, we focus on the specific problem of *location-dependent services*, i.e., context-aware services whose execution can be dynamically adapted depending on the current location of users. We emphasize that our concept of location is very general, encompassing both physical location (e.g., a user located in a specific street in a town or in a specific room in a building), and "logical" location. The latter refers to the fact that the activities of a user may be logically located within a specific distributed group/application (e.g., a user located in a team of coworkers, or in the administrators' group of a specific information system). For simplicity's sake, we will refer to "*location-dependent services*" when related to physical location in space, and to "*group-dependent services*" when related to the logical location of users within a specific application group. As an additional note, we outline that the types of M-services we consider are not necessarily accessed by mobile users only, but by mobile software agents too. This is because we will all increasingly delegate autonomous software agents to access and use electronic resources. Thus, future M-services scenarios will blur the distinction between human and software agents. Accordingly, we use the term "mobile agents" to specify both mobile users and mobile software agents [9], [17], [19], and the term "M-service" to indicate services being accessed by both mobile users and mobile software agents. From a software engineering point of view, in fact, the challenges to adapt M-services to the agents' context are the same whether we consider mobile users or mobile software agents.

Given this background, the contribution of this paper is twofold. First, we propose a new modeling framework for the design and development of location-dependent M-services and, more generally, of distributed agent-based mobile applications using such services. Second, we show how an appropriate middleware infrastructure can be exploited to actually implement the general concepts of the framework and to deploy applications and services accordingly.

Manuscript received August 15, 2003. This work was supported by the Italian MIUR and CNR in the Progetto Strategico IS-MANET, Infrastructures for Mobile Ad-Hoc Networks. This paper was recommend by Guest Editors B. Benatallah and Z. Maamar.

The authors are with University of Modena and Reggio Emilia, 41100 Modena, Italy (e-mail: cabri.giacomo@unimo.it; leonardi.letizia@unimo.it; mamei.marco@unimo.it; zambonelli.franco@unimo.it).

Digital Object Identifier 10.1109/TSMCA.2003.819496

With regard to the first contribution, we start from the considerations that, to deal with mobility and context awareness, it is necessary to have a modeling framework that takes these concepts as first-class entities.

Therefore, we propose modeling the agents' operational environment as a set of *local service contexts*, each representing the "place" in which agents' activities are situated. For example, we could model a building by identifying each building's room as a local service context. An agent, moving across the building, would enter and leave different rooms (i.e., local service context) and its actions would be performed in a specific local service context depending on the room in which it is in.

Each local service context defines a locality scope within which agents can access the locally available M-services. In the example before, a number of M-services would be deployed on various rooms and a M-service installed in room would not be accessible from another room.

Accordingly, an M-service is not generally accessible from *everywhere*, but only from where it is meaningfully located and/or replicated.

This leads to an explicitly modeling of (physical) location-dependencies in service provisioning. For example, consider an M-service providing a city-wide yellow-pages facility. In our approach, rather than a single globally accessible yellow-pages facility, the idea is to have a multiplicity of facilities, each deployed on a different area of the city and providing information restricted to that specific area (in our terminology, each of these areas would constitute a local service context). In this way, a query for restaurants would be restricted to restaurants in the neighborhood, that is, to restaurants belonging to the local service context accessed by the user. Moreover, in our framework, service contexts are not simply passive repositories of service. Rather, they are *active and reconfigurable contexts*, in charge of actively mediating and customizing the execution of services depending on the specific types of agents that are accessing them. Therefore, the same M-service on the service contexts could exhibit different executions if accessed by agents of different types (or belonging to different groups/applications). In other words, active service contexts lead to an explicit enforcement of logical location-dependencies (i.e., group dependencies).

With regard to the second contribution of this paper, it is rather clear that the potentials of our framework can be fully realized via the availability of a proper middleware infrastructure. In this paper, we show that the mobile agent reactive spaces (MARS) coordination infrastructure [8], by implementing the concept of local and active service contexts in terms of programmable tuple spaces, well supports the development and deployment of location-dependent M-services and of distributed applications exploiting them.

To clarify the concepts expressed and the effectiveness of our approach, we introduce an application example to act as a running case study through the paper. The example relates to a scenario of agent-mediated accesses to a distributed set of tourist information and services, with the goal of helping groups of tourists in planning their journey [5].

The remainder of this paper is organized as follow. Section II details the key concepts of our framework, discusses their impact on application design, and introduces the case study application. Section III presents the MARS coordination infrastructure and shows how it enables the definition and implementation of service contexts. Section IV shows how to exploit the MARS infrastructure to implement a variety of location-dependent services in the case study scenario. Section V discusses related work in the area. Section VI concludes the paper and sketches future works.

II. Service Contexts Framework

In this section, we introduce the general scenario of our proposal and the basic background concepts and definitions related to location-dependent M-services. Then, we detail the modeling framework based on local and active service contexts and discuss the impact of the framework on application design. Finally, we introduce the case study application and show how our framework can be applied to it.

A. Location-Dependent M-Services

The general scenario in which our proposal situates is that of a variety of mobile agents accessing data and services on a fixed network infrastructure. On the one hand, such agents could be nomadic humans exploiting some sorts of mobile devices (e.g., a PDA) to connect to the Internet either for work or for pleasure. These includes, for instance, employees of a company in need of accessing on-line information and services, as well as tourists visiting a town and wishing to access information about local cultural heritage [5]. On the other hand, such agents could be software agents, in charge of roaming in the Internet to retrieve information and perform actions on behalf of a user. These include, for instance, agents to collect and organize a set of distributed information, as well as agents in charge of accessing e-commerce facilities in agent-enabled e-marketplaces [23].

In the above scenarios, agents (whether human or software) are not necessarily stand-alone entities. Instead, their activities may take place in the context of a *multiagent system* [19], where an agent does not act alone, but works together and has to coordinate its actions with other agents. For instance, a nomadic worker could be in need of coordinating his/her actions with colleagues according to specific workflow rules [13]. A mobile agent looking for information may be in need of sharing partial findings with other agents performing a similar search in parallel with it [7].

In the above scenario, we consider M-services as the means to *enable the interactions* between mobile agents and network resources, as well as between different agents. To this purpose, we explicitly distinguish between *resource M-services* and *coordination M-services.*

— *Resource M-services* are all those services that enable the access to some kind of resource on the fixed network infrastructure. These include all types of traditional services giving access to data and information (e.g., data files and Web pages) as well as those services wrapping access to some kind of software application [e.g., a data base management system (DBMS)] or hardware devices (e.g., a printer).

— *Coordination M-services* are all those services that are conceived with the goal of supporting some sorts of inter-agent interactions. Instead of wrapping resources, coordination M-services wrap access to some communication and coordination media. Of course, a coordination M-service could be implemented in various ways, providing different communication and coordination models. For example, it could be realized by means of a tuple-space providing agents with a shared space upon which to exchange information and communicate [18], or as an event-based engine enabling interactions according to a publish-subscribe model [13].

In general, the above two classes cover most of the significant types of services that one could wish to exploit in a mobile setting.

Whatever resource or coordination M-services are involved, we have already stated in the introduction that the effective fruition of M-services requires *context-awareness* to promote *context-dependency*. Generally speaking, the *context* of a service identifies the operational environment in which the invocation of a service occurs. This may include, among the others, the hardware device from which it is invoked, the time of invocation, the physical location of the invoking agent, and the possible membership of the invoking agent to a specific multiagent system or group. Contextual information can be exploited so as to adapt the execution of services depending on the context from which they are requested (or on some specific characteristics of the context). Just to give a few examples: a multimedia service could lead to different renderings when invoked on devices with different display capabilities; a streaming service could provide different quality of services depending on the bandwidth available for transmission; a game server could adapt the difficulty of a game and the content of the scenery depending on the age of the player.

Although a variety of specific context-dependencies can be fruitfully promoted in M-services, we specifically focus on *location dependencies*, which we consider of a paramount importance in mobile scenarios. In particular, as anticipated in the introduction, we consider location dependencies as they relate to both physical location in space and logical location in a group/application (i.e., group membership).

— *Agent location*. The location of an agent trivially defines from where the agent accesses the M-service. Of course, physical location may encompass different granularities depending on the application needs and the available technologies. With regard to human agents, one can consider both raw geographical location [e.g., as provided by a global positioning system (GPS)] and information bounded to the actual operational environment (e.g., a specific room in a building or a specific street in a town). With regard to software agents, one typically considers location of an agent in terms of its current position on a specific node/domain of the network.

— *Agent group membership*. The fact that an agent may perform its activities in the context of a specific multi-agent systems or a logically correlated group of agent can be regarded as a specific form of logical location. With regard to human agents, group membership typically relates to some sort of social or organizational relations. With regard to software agents, group membership typically relates to the fact that they are created and execute in the context of a common distributed application.

On the basis of the above two types of contextual information, the execution of M-services can imply two different forms of dependency.

— *Location-dependencies*. The execution of M-services, and the very availability of an M-service, may depend on the specific location from which they are invoked. For example, a resource M-service wrapping a company database can impose different constraints depending on whether it is accessed from the boss office or from the coffee-break room. Similarly, a coordination M-service managing document co-editing can provide different policies to modify a document depending on whether it is accessed from a company office or by nomadic user outdoor.

— *Group-dependencies*. The execution and availability of M-services may depend on the group/application to which agents belong. For example, a resource M-service wrapping a database can allow free access to agents belonging to the administrator group, while imposing strict security constraints to other agents. A coordination M-service wrapping a chat server can impose different conversation rules for agents belonging to different interest groups, e.g., enabling concurrent conversations in recreational groups, while imposing a reservation-based conversation model in professional groups.

We are aware that location and group dependencies in M-services may not cover all possible types of context-dependencies that one can meaningful enforce in M-services. However, we consider them so important to require a suitable framework to model M-services and distributed applications around them. How such framework could also deal with other types of contextual dependencies will anyway be discussed later in the paper.

B. Service Contexts

The modeling framework we propose aims at promoting the modular design of location-dependent and group-dependent M-services. More generally, it is intended to support the integrated design of M-services and of distributed multiagent applications accessing them to carry on their activities and to coordinate with each other.

Our modeling framework takes the concepts of mobility and context-awareness as first-class entities, and promotes a uniform modeling of the different types of M-services (resource and coordination ones) and of the different types of contextual dependencies (location and group ones).

The central abstraction of the framework is *service context*. A service context is the logical abstraction of the place in which an agent executes, i.e., the place in which its activities situate and in which it has the chance of accessing a set of locally available

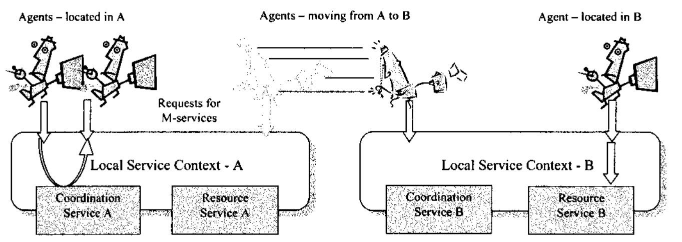


Fig. 1. Service context scenario. Agents move across local service contexts. In each service context, they can access locally available services, whether resource or coordination ones.

services (see Fig. 1). These services may include both resource M-services and coordination M-services. Depending on the actual application scenario, a service context can be used to model a network node, an administrative network domain, or a wireless access point.

A distributed network environment is logically modeled in terms of a set of localized and independent service contexts. Service context are defined, within an application scenario, to meaningfully represent a homogeneous and logically bounded piece of the environment. For example, in an application deployed in a building, they could coincide with the building's rooms, while in a city-wide application, each local service context could represent a district.

Agent mobility is logically modeled in terms of changes in the service context in which an agent situates. Thus, the general scenario is that of a number of agents, spending their nomadic lives from a service context to another, and having the possibility of accessing, at a given moment of their lives, the services available by their current service context.

The abstraction of local service contexts reflects—at the level of services and applications modeling—a notion of context intrinsic in mobility. In fact, for the very fact of moving across different environments, agents access different data and services, and interact with other agents, depending on their location. As stated in the introduction, this means that, in our perspective, there are not globally accessible M-services. Instead, M-services are actually localized within a local service context and accessible only by locally executing agents. Of course, it is not our intention to exclude the possibility of promoting the modeling of globally accessible services. However, in our framework, they would have to be considered in terms of a number of replicated (or coordinated) localized M-services. Consequently, the access to M-services in the framework is intrinsically location-dependent. A service may exists and be available in a location, while it may not exist (thus, not accessible) in others.

Service context, in our model, are not simply passive repositories of data and services. Instead, they are considered active entities, in charge of mediating all locally occurring interactions. These include interactions between local agents and local resource M-services, as well as interagent interactions occurring via some coordination M-services. Moreover, service contexts are considered reconfigurable entities, enabling the dynamic adaptation of services' execution.

With regard to the latter point, we consider that the execution of each local service context can be independently programmed to configure the *rules* according to which specific

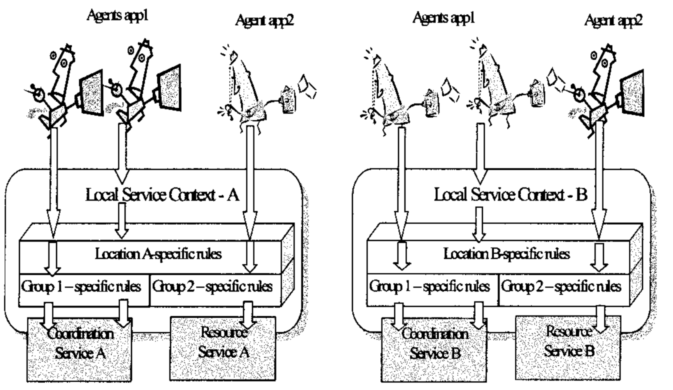


Fig. 2. General framework based on local and active service contexts.

services should be accessed. Accordingly to the identified two types of dependencies (location- and group-dependencies), two different kinds of rules can be considered.

— *Location-specific rules*. These rules are local to a service context, and are intended to apply to the agents executing in that context to adapt the execution of service to the specific characteristics of the local context.
— *Group-specific rules*. These rules are intended to be associated with all the agents belonging to a specific group/applications. When an agent of a specific group accesses services on a local service context, the local service context adapts its execution via the enforcement of rules applying to all the agents of that group, and only to them.

Putting all together, the introduced concepts lead to the scenario depicted in Fig. 2. This represents a usable and modular conceptual framework for the analysis and design of distributed mobile applications accessing M-services. On the one hand, different service contexts may enact different *location-specific rules*. These location-specific rules integrate and extend the concept of location-dependency that is already intrinsic in local service context. The services available to an agent intrinsically depend on the local service context in which it is executing. In addition, the enactment of location-specific rules can lead to differentiated executions in similar services available on different services contexts: the same service in different sites may exhibit different executions in response to an agent request. On the other hand, different *group-specific rules* can be enacted for all the agents of a group on any visited site. Thus, the same service on the same service context can lead to differentiated execution depending on who is accessing it. Location-specific rules can act concurrently with the group-specific ones, so that the resulting execution of a service derived from adaptation to both types of rules.

All types of above rules can be preinstalled in a service context (e.g., by the local administrator) or dynamically installed by agents themselves accordingly to their specific needs.

C. Designing Applications Around Service Contexts

The framework introduced above defines useful abstractions to guide and simplify the process of developing distributed multiagent applications accessing M-services. In fact, the model naturally invites in designing an application by clearly separating the *intraagent* issues—related to the specific internal computational activities of agents—and the *interagent* ones—related to the interaction of the agents with M-services (both resource and coordination ones). In other words, the framework promotes a clear separation of concerns, which is likely to reduce the design complexity of both applications and services.

The definition of a detailed methodology is outside the scope of this paper. Still, we can sketch some general guidelines for applications and services design, based on the identification of intraagent and interagent issues. It is worth noting that these guidelines are coherent with the ones already identified in the area of agent-oriented software engineering and methodologies [33]–[35].

From the point of view of application developers, the design of an application can be organized as follows:

— at the intraagent level, one has to analyze the application goal and, in the case of a multiagent application, decompose it into subgoals. This process should lead to the identification of the agents that will be instantiated in the application (or of the software components that a user will exploit to perform its required nomadic activities) [33]. Relevant characteristics to be identified for each agent may include both functional ones (necessary for the achievement of a specific sub-goal) and nonfunctional ones (e.g., a proper internal agent architecture);

— at the inter-agent level, one should identify what are the suitable M-services that have to be accessed by the agents to let them achieve a global application goal [35]. In particular, this amounts at identifying both the services used by agents to interact with each other (*coordination M-services*) as well as those services used to access resources (*resource M-services*). Moreover, this implies defining the activities that service contexts must enforce to properly mediate and support the execution of the above agents to M-services interactions. These supporting activities will be expressed by means of the previously introduced *location* and *application-specific rules*.

Our framework explicitly assigns specified duties also to the *site administrators* of local service contexts, i.e., to those persons in charge of managing and configuring the M-services offered in local service contexts. When new kinds of application agents are known to be deployed on the distributed scenario, the administrator of one site can identify all the local, location-dependent, rules that (s)he may find necessary to enact for the execution of the application agents in the site. These rules can be used to facilitate the execution of the agents on the site, and/or to make the structure of the local organization homogeneous to agents' expectations, and/or to protect the site from improper exploitation of the local contexts. In general, the identification of the location-dependent rules is intended to define what local policies should be defined to govern the accesses to the local services and to locally tune the execution of services. Our framework supports directly site administrator by providing means to enforce the above specified rules.

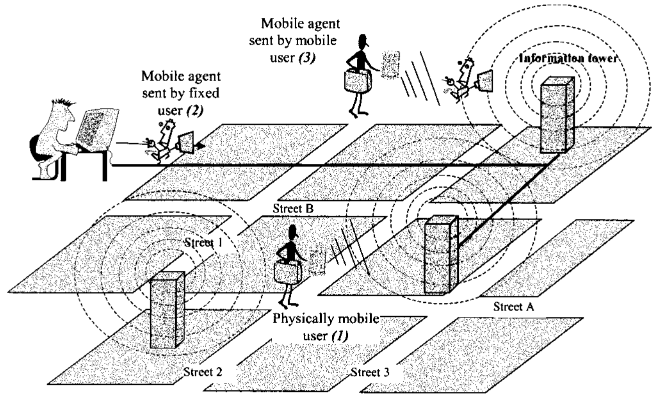

Mobile agent sent by mobile user *(3)*
Information tower
Mobile agent sent by fixed user *(2)*
Street B
Street 1
Street A
Physically mobile user *(1)*
Street 2
Street 3

Fig. 3. Scenario of the case study.

D. *Case Study: Distributed Information Towers*

Tourism is one of the fields that most notably will take advantage of modern information and communication technologies. Portable devices, wireless access to data and services, personal agents and mobile agents for information retrieval, can altogether be fruitfully exploited to enrich tourists' experiences [5].

As a specific case study, we focus on the problem of accessing the information and services related to the tourist resources of a city, with the goal of properly organizing a visit on-the-fly. We assume that the city is provided with a suitable distributed communication infrastructure. In particular, let us assume that the city is furnished with "information towers," distributed in the city and providing location-specific information and services related to the local region of the city in which they are situated (see Fig. 3). As an example, should such an infrastructure be found in Rome (Italy), then an information tower by Città del Vaticano would provide information about visiting the Vatican museums and the Cappella Sistina; an information tower by Trastevere would provide information about the Santa Maria Church and services to reserve a table in a typical Trastevere restaurant. These information towers clearly reify the abstraction of local service contexts.

We suppose that the access to information and to services on the information towers is provided via different means. First, information towers are enabled to provide information and services via wireless connections (see case 1, in Fig. 3). So, tourists that are actually visiting the city by moving around in its streets can exploit a personal digital assistant running on a palm computer or smart cell-phone to access local information towers and there discover what is around. For instance, a person walking in Trastevere can connect to the local information tower to discover if there is a Chinese restaurant in the neighborhood, and possibly to reserve a table.

Moreover, information towers are supposed to be connected to the Internet, so that they can host the execution of mobile software agents. Users can thus send their personal assistant mobile agents to roam across the information towers and there access information and services. This facility could be used to send mobile agents to organize an off-line visit of the city, prior to the actual one (see case 2, in Fig. 3). Also, once the tourist is in place, this facility could be used to deploy a mobile agent in the network to access remote information towers without having the tourist to actually walk there (see case 3, in Fig. 3).

The above scenario can be considered as representative of several other application areas, such as intelligent homes, traffic information systems, or intelligent museums. In fact, all these application areas are characterized by the presence of a fixed infrastructure based on a multiplicity of nodes, each associated to a specific location and providing location-sensitive information and services. For all these cases, the modeling of applications in terms of agents moving across local service contexts is undoubtedly the most suitable one. On the one hand, any type of "information towers" providing information and services can be naturally modeled in terms of local service contexts in which to access M-services. On the other hand, any software component accessing information and exploiting services can be modeled in terms of an agent that moves across different service contexts.

E. Service Contexts in the Case Study

Let us consider the problem of designing an agent-based application in charge of properly organizing a city tour for a tourist or for a group of tourists. The tourists may have specific preferences, time constraints, and budget limitations. The application should be able of organizing a tour of the city accordingly to such requirements.

At the intraagent level, one has to decide whether and according to which criterion to subdivide the application goal into subgoals, to be assigned to different agents. For instance, one could think at one kind of agent in charge of finding and selecting a set of interesting sites to be visited ("visit agent"). Another kind of agent could be in charge of looking for and reserving tables in restaurants ("reservation agent"), accordingly to both users' preferences and the decision of the visit agents. On the basis of the goals to be achieved by the identified agents, one can then decide what internal architecture for agents (e.g., object-based or a goal-oriented) is better suited.

At the interagent level, one has to analyze which M-services the application agents should use and which rules should be enacted by local service contexts (i.e., city information towers).

The types of M-services application agents may need to access in each local service contexts include the following:

— A resource M-service to retrieve, from an information tower, the location-dependent information needed to successfully organize a visit. This may include, e.g., the visiting time and admittance prices for local museums, bus schedules, local restaurant information;

— A coordination M-service to coordinate with other agents of the same application, e.g., the visit agent will have to coordinate its decisions with the reservation agent;

— A coordination M-service to interact with foreign agents, e.g., visit agents of different groups may need to coordinate to obtain a collective discount on the visit price to a site.

With reference to the first point, let us suppose that each information tower offers a *resource M-service* providing access to restaurant information in terms of "type of restaurant" only (e.g., Italian, Indian, Chinese, etc.). However, it may be the case that the reservation agent would like to have access to this information in terms of "class of restaurant" (e.g., cheap, expensive, luxury). To solve this mismatch, the trivial and ineffective solution is to force the reservation agent retrieve information about all restaurants of all types, and then select all the information retrieved on the basis of price criterion. The alternative and effective solution, enabled by the fact that service contexts are active and reconfigurable, can rely on a *group-specific rule* to "adapt" the execution of the accessed service context to its own need. In particular, this implies integrating in service contexts a rule devoted to accept requests for the restaurant information in terms of restaurant classes, and handle them accordingly. The advantage, in this case, is that one can change the group-specific rule to change the way restaurants are looked for by changing neither the agent code nor the service.

With reference to the latter two points, let us abstract from the specific coordination M-service that the agents colocated on a local service contexts can exploit to interact, and simply assume there is one.

Whatever the coordination M-service, identifying that a visit agent and a reservation agent will have to coordinate together raises a question about how this coordination process should be actually ruled. As one possibility, the group of tourists can decide that the visit *per se* is more important than what to eat during the visit. Thus, it may require that a restaurant agent can book a reservation only after the schedule of the visit has been decided. This implies defining a *group-specific rule* preventing the reservation agent to proactively initiate any activity before the visit agent has completed its task. Conversely, the group could give gastronomy more importance, by organizing the visit on the basis of the selected restaurants. Thus, it has to rule the interactions between the visit and the reservation agent, via a proper group-specific rule, so that the former can start its activity once the latter has completed. Again, the fact that the two different solutions are enforced in term of a group-specific rule makes it possible to switch from one choice to the other by having to change neither the agents' code nor the coordination service.

Let us describe another example, related to the interaction among agents of different groups. One can consider a situation in which different visit agents are willing to interact to organize a larger group on-the-fly, so as to obtain a significant discount on the access price to a given historical site. Also in this case, one can think at a *group-specific rule* that caches the booking requests for a given tour issued by single agents, and releases a larger group tour once the critical mass is reached.

Separated from the above design issues, the perspective of the local administrators is to enact *location-specific rules* on the information towers, to control the accesses and the coordination activities of application agents. As a simple example, the administrator can create a location-specific rule that denies a reservation agent to initiate a protocol for the reservation of a restaurant if it has already reserved a place in another restaurant of the same local service context during the same time span. As another example, the administrator of a local context in a very attractive location could enact a rule requiring some sort of advance payment (e.g., via credit card charge) before confirming a reservation in a restaurant.

In all the above situations, the strength of our approach is to promote a clear separation of concerns. Such separation of concerns promotes adapting the overall execution of application agents and services to contingent needs without having to change agents and M-services. Most adaptation needs can be accommodated by means of groups-specific and location-specific rules being enacted in local service contexts.

III. MARS Infrastructure

MARS, developed at the University of Modena and Reggio Emilia and described in detail in [8]), is a software infrastructure that directly maps the concept of local service contexts into a distributed middleware architecture based on *programmable coordination media*.

In general terms, a *coordination medium* is a software system that mediates interaction and coordination among a set of agents executing within a locality [14]. To do that, a coordination medium relies on a specific and uniform interaction model, according to which all interactions will take place (both accesses to local services and interagent interactions). Message-oriented middleware [27], event-based engines [13], tuples spaces [18], virtual data structures [26], are all examples of coordination media relying on different interaction models. *Programmability* of a coordination medium stems from the possibility of programming its internal behavior in response to access events [8], [14], [27]. To grant transparency, the capability of programming the behavior has to be made available without changing the set of primitive interaction operations used to access the coordination medium.

Programmable coordination media, in general, translate the abstractions of local service contexts into a set of concrete software entities and into a set of actual APIs. Therefore, an infrastructure based on distributed programmable coordination media enables to preserve the separation of concerns between intraagent issues and inter-agent (as discussed in Section II) also during the development and the maintenance. In fact, if the code of the agents can be clearly separated from the code needed to program the behavior of coordination media (i.e., the code implementing *location-specific* and *group-specific rules*), agents and rules can be coded, changed, and re-used, independently of each other. Thus, by avoiding to hardwire into agents the code related to the implementation of specific coordination rules, such an infrastructure can promote the writing of modular, manageable, and reusable code.

A. MARS Architecture and Interface

MARS implements programmable coordination media in terms of programmable tuple spaces, with an interface compliant with that of Sun's *JavaSpaces* [16]. A tuple space is a shared repository of information, which can be accessed via well-defined primitives; a tuple space can be considered as a blackboard with a mechanism that permits to retrieve partially known information (see the description of the pattern-matching in the following). Globally, the MARS architecture is made up of a multiplicity of independent programmable tuple spaces. One tuple space is typically associated to a node, and it represents the local service context for that node (see Fig. 4).

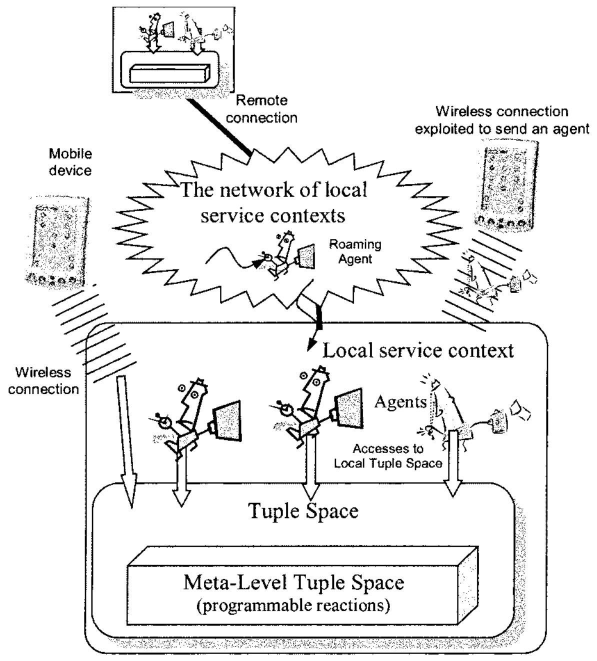

Remote connection
Wireless connection exploited to send an agent
Mobile device
The network of local service contexts
Roaming Agent
Wireless connection
Local service context
Agents
Accesses to Local Tuple Space
Tuple Space
Meta-Level Tuple Space (programmable reactions)

Fig. 4. MARS architecture.

Agents are supposed to access a MARS tuple space via a private reference, which is bound to the MARS tuple space representing the current local-service context. The binding of the private reference to an actual tuple space automatically changes on the basis of the agents' movement.

Accordingly to the general mean of "agent" introduced in the Section I, MARS supports the access by different kinds of software components, either executing local to the accessed tuple space or executing on a remote device. In the latter case, a software agent running on a mobile computing device is able to catch "connection events," i.e., those events generated by the wireless-enabled nodes of the fixed network infrastructure as soon as a device enters its connection range. Then, the solution adopted in MARS is to deliver the reference of the local MARS tuple space to a connecting device together with the connection event itself. The agent, by its side, should handle this event at its willing, typically by binding a private reference to the MARS tuple space it is connecting to. The current implementation supports IEEE 802.11 and infrared technologies, even if it can easily be adapted to other connection technologies, such as Bluetooth. It is worth noting that this process has been made robust — by means of periodic messages' exchange—to account for wireless intermittent links.

In MARS, tuples (usually called "entries") are Java objects whose instance variables represent the tuple fields. The entry objects must be instances of classes implementing the `Entry` interface, but usually they derive from the `AbstractEntry` class. These tuple can also be used to define template tuples, by leaving the values of some variables undefined (`null`), to be used in the pattern matching mechanism described later.

To access the tuple space, the write, read, and take operations are provided to store, read, and extract, respectively, a tuple, based on a template tuple. In addition, the `readAll` and `takeAll` operations are provided to read and extract, respec-

```
public interface MARS extends JavaSpace {
    // method interface inherited from JavaSpace
// put a tuple into the space
// Lease write(Entry e, Transaction txn, long lease);
// read a matching tuple from the space
// Entry read(Entry tmpl, Transaction txn, long timeout);
// extract a matching tuple from the space
// Entry take(Entry tmpl, Transaction txn, long timeout);

// added methods
// read all matching tuples
Vector readAll(Entry tmpl, Transaction txn, long
timeout);
// extract all matching tuples
Vector takeAll(Entry tmpl, Transaction txn, long
timeout);
}
```

Fig. 5. MARS interface.

tively, all matching tuples from the space. Fig. 5 reports the Java interface of MARS tuple spaces. As can be seen, all operations can be part of a transaction, to allow atomic series of operation and mechanisms of rollback. Moreover, the retrieving operations allow the specification of a timeout: the operations are blocked until it expires, after that they return even if no tuples have been found. Instead, the write operation permits to specify the time to live (lease) of the written tuple, after which the tuple is deleted from the tuple space. In the following, we disregard transaction and lease issues; interested readers can find more information in [16].

The `read/take` and `write` operations can be used also to synchronize agents. In fact, an agent performing a blocking `read/take` operation of a nonexisting tuple is suspended until another agent writes such tuple (or a matching one). This simple synchronization mechanism is at the base of the coordination mechanisms of tuple spaces [1].

The default behavior of a MARS tuple space in response to access events is a quite traditional pattern-matching access to tuples. In such a pattern-matching, as in Javaspaces, a template tuple *TMPL* and a tuple T match if and only if

— *TMPL* is an instance of either the class of T or of one of its superclasses; this extends the Linda model [1] according to object-orientation by permitting a match also between two tuples of different types, provided they belong to the same class hierarchy;

— the defined fields of *TMPL* that represent primitive types (integer, character, Boolean, etc.) have the same value of the corresponding fields in T;

— the defined nonprimitive fields (i.e., objects) of *TMPL* are equal—in their serialized form—to the corresponding ones of T (two Java objects assume the same serialized form only if each of their instance variables have equal values, recursively including enclosed objects).

For instance, the template (`null`, 10, `null`) matches with the tuples (“`ciao`,” 10, `true`), (`null`, 10, `false`) and (“`foo`,” 10, `null`), but not with the tuple (“`Italy`,” 11, `null`). A template tuple *TMPL* has to be provided as parameter of `read`, `take`, `readAll` and `takeAll` operations.

```
class RstTuple extends AbstractEntry {
  // AbstractEntry: generic tuple
  public String Type; // the type of the restaurant
  public String Name; // the name of the restaurant
  public String Address; // the address of the restaurant
  public Integer Price; // price class of restaurant
  // constructor
  public RstTuple(String t, String n, String a, Integer p) {
    Type = t; Name = n; Address = a; Price = p;
  }
}

class CustomerAgent extends Agent {
  // an agent that retrieves information about restaurants
  private Space TS; // reference to MARS tuple space
  ...
  // create a template specifying only the restaurant type
  RstTuple tmpl = new RstTuple("Italian", null, null, null);
  // retrieve all Italian restaurants
  Vector tv = TS.readAll(tmpl, null, NO_WAIT);
  // elaborate the retrieved vector
  ...
  // refine the template
  tmpl.Address = "Via del Corso";
  // require a restaurant in a specific street
  RstTuple b = (RstTuple) TS.read(tmpl, null,
NO_WAIT);
  ....
}
```

Fig. 6. Example of use of MARS.

Fig. 6 reports an example of code related to the case study, in order to show the use of MARS tuple spaces; this example is quite simple, and its purpose is to show how the MARS operations can be exploited. Let us consider a tuple space that contains information about restaurants. To this purpose we define the class `RstTuple` that implements tuples in the form (Type of `Restaurant`, `Name`, `Address`, `AveragePrice`).

The code of the `RstTuple` class is reported in the first half of Fig. 6. The tuple space contains one of these tuples for each restaurant. In this simple example, consider the case of a very simple `CustomerAgent` that accesses the tuple space to retrieve information about local restaurants. Such an agent acts in behalf of a tourist that is interested in having lunch in one restaurant in the area served by the tuple space—for instance, a city or a part of it. The agent is an instance of the `CustomerAgent` class, which is reported in the second half of Fig. 6. It retrieves information about restaurants from the tuple space TS of class `Space` that implements the MARS interface: by the `readAll` operation it retrieves information about all Italian restaurants, and then can elaborate the returned results; by the read, operation, instead it retrieves information about an Italian restaurant located in “Via del Corso” (one of the main streets of Rome). In both operations the timeout parameter is set to NO_WAIT, which means that the operation is nonblocking, returning `null` if no matching tuple is found. In fact, in this case a `CustomerAgent` cannot wait until a restaurant with the needed features will be built.

B. MARS Programmable Model

The programmable model of MARS enables the association of any needed reactions in response to access events performed by agents. In this way, a reaction can be exploited both to code a

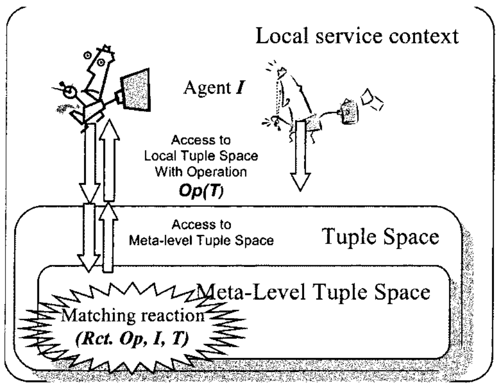

Local service context
Agent *I*
Access to Local Tuple Space With Operation *Op(T)*
Access to Meta-level Tuple Space
Tuple Space
Meta-Level Tuple Space
Matching reaction (*Rct. Op, I, T*)

Fig. 7. MARS reactive behavior.

service *per se* (overcoming the limitation of the basic tuple space model, which is basically data-oriented rather than service-oriented) as well as to code location-specific and group-specific rules.

The association of reactions to access events occurs via *4-ples* stored in a "*meta-level*" tuple space. A meta-level 4-ple has the form of *(Rct, I, Op, T)*: it means that the reaction method (representing the reaction itself) of an instance of the class *Rct* has to be triggered when an agent with identity *I* invokes the operation *Op* on a tuple/template T. Putting and extracting tuples from one meta-level tuple space corresponds to installing and uninstalling, respectively, reactions associated to events at the base-level tuple space. An administrator can do that via special-purpose agents or via a simple graphical user interface (GUI). Agents can do that via event-handlers that install the specified reactions automatically, as soon as they get connected to a tuple space, and without interfering with the agents' activities. We emphasize that the identity *I* of an agent can refer both to a unique agent identifier as well as to a group/application identifier.

Fig. 7 shows how an access by an agent may trigger a reaction in the meta-level tuple space, depending on: the identity of the agent (or of its group); the performed access; and the involved tuples. The reaction can modify the behavior of the tuple space in the sense that it can both manipulate the involved tuples as well as access the tuple space to perform further operations.

A meta-level 4-ple can have some nondefined values, in which case it associates the specified reaction to all the access events that match it. For example, the 4-ple (`Reaction`, `null`, `read`, `null`) in the meta-level tuple space associates the reaction of an object instantiated by the `Reaction` class to all `readoperations`, whatever the identity of the performing agent and the template tuple type and content. When an access event to the base-level tuple space occurs, MARS issues a special pattern-matching mechanism in the meta-level tuple space to look for reactions to be executed in response to the access event. If several 4-ples satisfy the meta-level matching mechanism, all the corresponding reactions are executed in a pipeline, accordingly to their installation order (or to a specific order determined by the administrator). When a reaction method executes, it is provided with parameters (detailed later) useful to characterize the access event that has triggered the reaction itself.

```
class PriceClass implements Reactivity
{
  // the delta that specifies the price range
  private static final int delta = 10;

  // the reaction method
  public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
  { //InputTuples contains the restaurants in the specified
   // price class, as from a normal readAll operation
    Vector v;
    int p = ((RstTuple)Template).Price.getValue();

    // retrieve restaurants in the higher price class
    ((RstTuple)Template).Price = new Integer(p+1);
    v = s.readAll(Template, null, NO_WAIT);
    // add the new tuples to the retrieved ones
    InputTuples = arrayMerge(InputTuples, v.toArray());

    // retrieve restaurants in the cheaper price class
    ((RstTuple)Template).Price = new Integer(p-1);
    v = s.readAll(Template, null, NO_WAIT);
    // add the new tuples to the retrieved ones
    InputTuples = arrayMerge(InputTuples, v.toArray());

    // restore the original price
    ((RstTuple)Template).Price = new Integer(p);

    // eventually return the retrieved tuples
    return InputTuples;
  }
}
```

Fig. 8. Example of MARS reaction.

Since reactions can be associated to access events either when performed by agents of a specific group or by any agent, they can be used to implement both *location-specific* rules and *group-specific* rules. On the one hand, depending on its own needs, the administrators can install reactions that apply to all the agents executing on their site, to enforce location-specific rules. On the other hand, agents can install their own group-specific reactions on the visited nodes, to apply only to agents belonging to the same group, to enforce group-specific rules. Specific security mechanisms in MARS guarantee that agents installing group specific rules, unless explicitly authorized, cannot influence the activities of agents in other groups.

Let us now exemplify the usage of MARS reaction. Referring to the previous example of the restaurants, let us consider an agent/application wishing to retrieve information about restaurants in a range of price classes. Such a service is not supported per se by a tuple space. In fact, a tuple space would require either to specify in the template a specific price class (and would return only those restaurant with that price class), or to specify a null value in the price class (and would return restaurants of any price class). To solve the problem, one could think at a reaction that, although accepting a template with a specific price class, is capable of returning to the invoking agents the list of restaurants whose price class differs from the specified one of a delta (plus or minus). Of course, such a reaction must influence only the accesses of the customer agents belonging to a specific group, because other groups may not be interested in (and would possibly deprecate) such a service. Therefore, it must be installed as a group-specific reaction.

Fig. 8 shows the code of the `PriceClass` class that roughly implements the above-described reaction. The core of the reaction is implemented by the reaction method, which is executed in response to an access. This reaction can be associated to `read` and `readAll` operations by inserting the 4-ples (`PriceClass`, `PriceApplicationID`, `read`, (`null`, `null`, `null`, `null`)) and (`PriceClass`, `PriceApplicationID`, `readAll`, (`null`, `null`, `null`, `null`)), respectively, in the meta-level tuple space. These 4-ples specifies that the `PriceClass` reaction will be triggered when an agent with the identity `PriceApplicationID` (intended as the identity of the group) performs a read or a `readAll` with whatever `RstTuple` template (the last sequence of null in the inner parenthesis).

Analyzing the reaction method of the `PriceClass` class, we can recognize the following parameters to the method:

— `Spaces`, a reference to the base-level tuple space to be used to let the reaction access the (base-level) tuple space.
— `IdentityId`, the identity of the agent that has triggered the reaction.
— `OperationOp`, the operation the agent has performed.
— `EntryTemplate`, the tuple provided as parameter in the operation.
— `EntryInputTuples`, the array of the tuples returned in the reaction previously executed in the pipeline, if any; otherwise, the tuples deriving from the normal pattern-matching mechanism.

In particular, in the `PriceClass` reaction, the `Template` parameter is used to keep the agent requirements, if any, the `s` parameter (as obvious) is used to retrieve tuples from the tuple space, and the `InputTuples` parameter reports the tuples retrieved so far. The reaction first performs a `readAll` operation to retrieve the restaurants in the more expensive price class (we assume price classes values as members an ordered type); then, a similar `readAll` operation is performed to retrieve the restaurants in the `cheaper` price class. In both cases, the results are merged with the tuples retrieved by the normal pattern-matching. Eventually, the array of all retrieved tuples is returned. Note that this reaction works well even in case of a single `read`, in which case only the first tuples of the array will be returned to the agent.

IV. Developing the Case Study in MARS

By assuming the presence of the MARS infrastructure in information towers, then

1) all information to be accessed by agents will be stored in the form of tuples in the tuple space;
2) services will be accessed by requesting proper tuples from the tuple spaces, triggering service executions;
3) all interaction between agents will assume the form of tuple exchanges and synchronization over tuple occurrences.

To develop the case study described in Section II by means of the MARS infrastructure, agents, M-services and location/group-specific rules are to be coded separately. The latter ones assume the form of MARS reactions to be installed into the meta-level tuple spaces either by application agents or by local administrators. In the following we will show how the service examples mentioned in Section II can be fruitfully implemented by means of MARS reactions.

```
// Excerpt for the class representing reservation agents
class ReservationAgent extends Agent {
private Space TS; // reference to MARS tuple space
public void run()   // active code of the agent
{...
RstTuple rstt, rsta[]; // tuples with restaurant information
RsvTuple rsvt; // tuple with request for reservation
AwTuple awt, awa; //tuples with confirmation

// the agent moves to a specific information tower
// this is a method that blocks the thread until a new
// information tower has been connected
this.waitUponConnection();


// creation of a template tuple  – class RstTuple
rstt = new RstTuple("Italian", null, "Via del Corso", null);
// a readAll operation is performed to retrieve information
// about all italian restaurants in Via del Corso
rsta[] = TS.readAll(rstt, null, NO_WAIT);
...
// internal activity to select a good restaurant
...
// creation of a reservation tuple
rsvt  =  new  RsvTuple(ChosenRestaurant,  4,  13.30,
MyIdentity);

// write a tuple requesting to reserve a table for 4 people
// at 1.30 pm at the ChosenRestaurant
TS.write(rsvt, null, 1000);

// template tuple for reading the answer tuple
awt  =  new  AwTuple(ChosenRestaurant,  4,  13.30,
MyIdentity, null);
// read the answer tuple, blocking until it is found
awa = TS.read(awt, null, 0);
if(awa.answer == true) // reservation confirmed
...// go on with the process ...
}
```

Fig. 9. Code of the reservation agent (fragment).

The first example (adapting the way restaurant information are provided to agents) has already been shown in Section III to explain the MARS programmable model.

With regard to the reservation service, Fig. 9 shows a (commented) fragment of the code of the reservation agents. An agent of this `class` accesses a local MARS space to retrieve information about `restaurants` (in a way similar to that of the simple `CustomerAgent` presented in Section III), and selects a `restaurant` on the basis of its needs. Then, the reservation agent writes a tuple in the space to express its willing of reserving a table in that restaurant. The writing of this tuple is supposed to trigger the execution of a service, possibly involving other agents, and eventually producing an “answer tuple” to confirm the reservation or not. The reservation agent, by its side, `after` having written the tuple, attempts to read a confirmation tuple and waits until it is produced (having specified 0 as timeout, meaning indefinite waiting time), thus, exploiting the read-write synchronization mechanism previously explained.

Fig. 10 shows the code of a reaction implementing the group-specific rule described in Section II, which avoids a reservation agent to reserve tables at a restaurant before the visit agent has completed its task. The reaction simply searches in the space for a tuple expressing the fact that the visit agent has completed its task. If such a tuple is found, it lets the reservation agent write

```
class WaitVisit implements Reactivity
{
 public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
  { // checks if the visit agent has already notified, via a
   // specific tuple, the fact that it has completed its duty
   VisitTuple va, vt = new VisitTuple(null, null, Identity);
   va = s.read(vt, null, 0);
   if(va == null)    // no found tuple, which means the
                     // visit agent has not completed its duty
   { // write an answer tuple to deny the reservation
     AwTuple at = new AwTuple ();
     at.Place = (RsvTuple)Template.Place;
     at.Persons = (RsvTuple)Template.Persons;
     at.Time = (RsvTuple)Template.Time;
     at.Identity = Id;
     at.answer = false; // negative answer
     s.write(at, null, 10000);
     return null // no tuple to be written in the space
   }
   else // the visit agent has completed its task
   { return InputTuples;
     // return the reservation tuple to be eventually
     // written in the space and let the process go on
    }
}}
```

Fig. 10. WaitVisit reaction class.

the reservation tuple in the space. Otherwise, it does not write the reservation tuple in the space: it immediately puts an answer tuple communicating to the reservation agent that its reservation has not been confirmed, and then returns `null` to prevent the reservation agent to write any reservation tuple in the space.

This reaction can be installed in MARS by means of the 4-ple (`WaitVisit`, `null`, `write`, (`null`, `null`, `null`, `null`)), where the last sequence of `null` matches a `RstTuple` with any values.

A very similar code can be exploited for implementing the location-dependent rule that avoids a reservation agent to reserve a table if it has already made another local reservation. This reaction is shown in Fig. 11, and it can be pipelined with the group-specific reaction of Fig. 10. In other words, in MARS, location- and group-specific rules can be enforced concurrently and in a harmless way.

As in the previous example, the reaction can be installed in MARS by writing the 4-ple [(`SingleReservation`, `null`, `write`, (`null`, `null`, `null`, `null`))], again matching any `RstTuple`.

We also emphasize that the separation of concerns enforced in MARS not only allows to change the code for agents and rules independently of each other, but also enables new services to be added and old ones to be dismissed influencing neither the behavior of agents nor the one of already programmed services. As an example, let us suppose the administrator of one site decides to change the local policy related to the management of multiple reservations: instead of denying a reservation, as in the previous case, the administrator decides that the new reservation can be considered and evaluated, while the previous one is to be cancelled. To enact this new policy, the administrator has simply to de-install the reaction of Fig. 11, and to install the new version of the reaction shown in Fig. 12, implementing the new policy. This change can be performed even at run-time, and influence neither agents nor the other services.

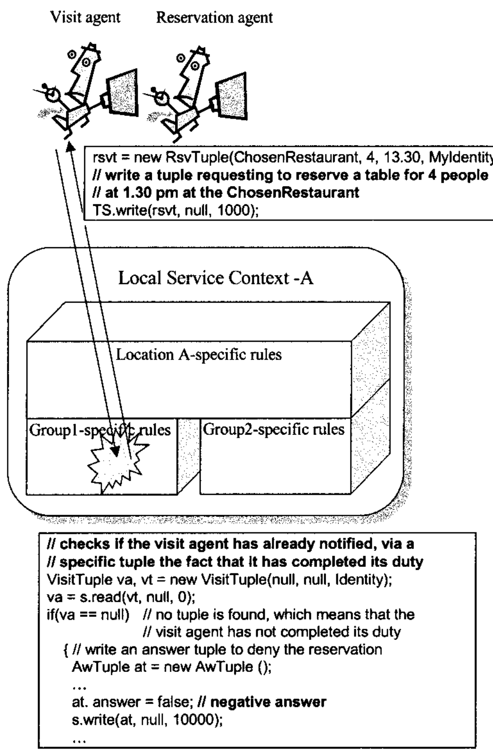
```
class SingleReservation implements Reactivity
{
 public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
  { RsvTuple rst;
   // read the time at which the reservation is made
   Time t = (RsvTuple)Template.time;

   // checks if another reservation for the same agent
   // and at the same time has been made
   rst = s.read(new RsvTuple(null, null, t, Identity, null),
null, NO_WAIT);

   if(rst != null)  // a previous reservation exists
   { // write an aswer tuple to deny the reservation
     AwTuple at = new AwTuple ();
     at.Place = (RsvTuple)Template.Place;
     at.Time = t;
     at.Persons = (RsvTuple)Template.Persons;
     at.Identity = Id;
     at. answer = false; // negative answer
     s.write(at, null, 10000);
     // return no tuple to be written in the space
     return null
   }  else // no previous reservation
     // return the reservation tuple to be written
     // in the space and let the process go on
     return InputTuples;
}}
```

Fig. 11. SingleReservation reaction class.

```
class SingleReservation2 implements Reactivity
{ // an alternative version of the same reaction

 public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
  { RsvTuple rst;
   // read the time at which the reservation is made
   Time t = (RsvTuple)Template.time;

   // checks if another reservation for the same agent
   // and at the same time has been made
   rst = s.read(new RsvTuple(null, null, t, Identity, null),
null, NO_WAIT);

   if(rst != null)  // a previous reservation exist
     // delete it from the tuple space
    s.take(rst, null, NO_WAIT);

   // return the reservation tuple
   // to be eventually written
   // in the space and let the process go on
   return InputTuples;
   }
}}
```

Fig. 12. Alternate version of the SingleReservation reaction class.

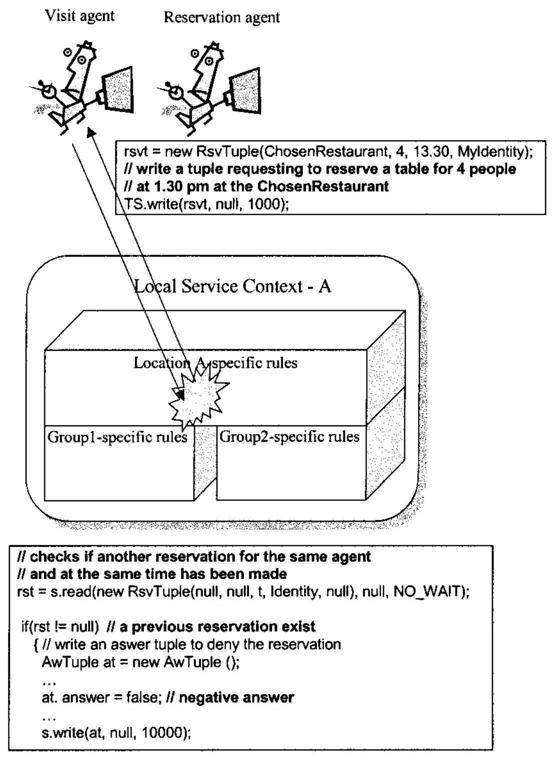
Figs. 13 and 14 graphically depict what happens in the cases of the first two previous reactions. Fig. 13 reports the situation in which the agent whose code is in Fig. 9 triggers the reaction of Fig. 10: the agent tries to reserve a table, but it is denied by a group-specific rule that avoids to perform a reservation until a visit has been scheduled. Fig. 14 is related to the reaction reported in Fig. 11: the agent tries to reserve a table, but it is denied by a location-specific rule that denies performing a reservation if the same agent has done another reservation.



```
rsvt = new RsvTuple(ChosenRestaurant, 4, 13.30, MyIdentity);
// write a tuple requesting to reserve a table for 4 people
// at 1.30 pm at the ChosenRestaurant
TS.write(rsvt, null, 1000);
```

```
// checks if the visit agent has already notified, via a
// specific tuple the fact that it has completed its duty
VisitTuple va, vt = new VisitTuple(null, null, Identity);
va = s.read(vt, null, 0);
if(va == null)  // no tuple is found, which means that the
                // visit agent has not completed its duty
  { // write an answer tuple to deny the reservation
    AwTuple at = new AwTuple ();
    ...
    at. answer = false; // negative answer
    s.write(at, null, 10000);
    ...
```

Fig. 13. Reservation agent triggers a WaitVisit reaction.



```
rsvt = new RsvTuple(ChosenRestaurant, 4, 13.30, MyIdentity);
// write a tuple requesting to reserve a table for 4 people
// at 1.30 pm at the ChosenRestaurant
TS.write(rsvt, null, 1000);
```

```
// checks if another reservation for the same agent
// and at the same time has been made
rst = s.read(new RsvTuple(null, null, t, Identity, null), null, NO_WAIT);

 if(rst != null) // a previous reservation exist
   { // write an aswer tuple to deny the reservation
     AwTuple at = new AwTuple ();
     ...
     at. answer = false; // negative answer
     ...
     s.write(at, null, 10000);
```

Fig. 14. Reservation agent triggers a SingleReservation reaction.

As a last example of coordination service, let us consider visit agents of different groups that wish to book a tour to a specific historical site. In this case, each visit agent of each group is in charge of notifying the tuple space about the willingness of booking the tour. Then, each agent has to wait for the beginning

```
class VisitAgent extends Agent {
 // now the visit agent deals with tours
 private Space TS; // reference to MARS tuple space
 ...
 // notify the interest in attending the tour
 // by putting a tuple in the tuple space
 TS.write(new BookingTuple("Colosseo"), null, 900000);
 // wait for the beginning of the tour by reading a tuple
 TS.read(new TourTuple("Colosseo"), null, 900000);
 ...
}
class Tour implements Reactivity
{ // the static variable that counts the bookings
 private static int count = 0;
 // the minimum number of participants
 private static final int N = ...;

 // the reaction method
 public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
 { // increment the counter
   count++;

   // if the minimum number has been reached
   if (count >= N)
   {
    // put a tuple to inform that the tour starts
    s.write(new TourTuple("Colosseo"), null, 1000);
    // delete all booking tuples
    s.takeAll(new     BookingTuple("Colosseo"),     null,
NO_WAIT);
   return InputTuples;
 }}
```

Fig. 15. Visit agent that deals with a tour (fragment).

of the tour itself. For instance, to notify the interest in attending the tour to the "Colosseo," a visit agent can put an appropriate BookingTuple in the tuple space as shown in Fig. 15, and then can wait for the confirmation of the beginning of the tour by performing a read of a tuple of type `TourTuple`. In this case, the `TourTuple` may not be present when the agent performs the `readoperation`, so it must decide how much time it can wait for. If the visit agent can wait 15 min as waiting time, it can set this time as timeout parameter of the `readoperation` (in milliseconds) and suspend itself until either a `TourTuple` is written in the tuple space or the timeout expires. The above is the general, "normal" case.

However, it can also be the case that a tour can start only when a minimum number of participants has been reached. In this case, it would be worthwhile for visit agents to wait not simply until a normal tour can be accommodated, but until a tour with a minimum number of partecipants becomes available. Without changing the code of visit agent, such a need can be easily implemented by means of a group-specific reaction. In fact, an appropriate reaction can be triggered whenever a `BookingTuple` is written in the tuple space, to keep count of the number of written `BookingTuples`. Then, the reaction can confirm a tour only when the total number of participants has been reached. This reaction is implemented by the `Tourclass` shown in Fig. 16, and is associated to the write operation by means of the meta-level 4-ple (`Tour`, `null`, `write`, ["`Colosseo`")], which means that the `Tourreaction` will be triggered when whatever agent (the `null` in the second field) performs a `write` operation with a `BookingTuple` specifying a visit to the Colosseo as the tuple to be written (the last field).

```
class Tour implements Reactivity
{ // the static variable that counts the bookings
  private static int count = 0;
  // the minimum number of participants
  private static final int N = ...;

  // the reaction method
  public Entry[ ] reaction(Space s, Identity Id, Operation
Op, Entry Template, Entry InputTuples [ ])
  { // increment the counter
    count++;

    // if the minimum number has been reached
    if (count >= N)
    {
      // put a tuple to inform that the tour starts
      s.write(new TourTuple("Colosseo"), null, 1000);
      // delete all booking tuples
      s.takeAll(new    BookingTuple("Colosseo"),    null,
NO_WAIT);
    return InputTuples;
}}
```

Fig. 16. Tour reaction class.

As a final remark, we emphasize we are aware that the strict locality model enforced by the service contexts framework and, consequently, by MARS, may be somewhat limiting for specific application problems. For instance, a restaurant reservation service (cf. the code examples of Figs. 11 and 12) could be better implemented as a globally coordinated service across a whole town. This could avoid a user to book, at the same time, different restaurants in different parts of the town. Currently, our framework and MARS can provide this via specific supporting mobile agents in charge of roaming in the network to coordinate services in different contexts. The identification of a more general solution, e.g., providing explicit modeling of interrelationships between service contexts, is being investigated.

V. Related Work

In the past few years, most of the researches in the area of M-services have focused on the very basic problem of *enabling* mobile access to services via portable devices [2], [11], [28]. Only recently researches in the area have started explicitly focusing on models and infrastructures for *adapting* mobile accesses, i.e., suited for the definition location-dependent and group-dependent services. With this regard, we can distinguish two main classes of approaches: *middleware*-based and *agent*-based ones. In the former class, all the logic of adaptation is integrated in a middleware layer. In the latter class, the idea is to delegate to an infrastructure of software agents the duty of providing adaptive mobile access to services. Our proposal characterizes mainly as a middleware-based one, although it shares some key points with agent-based ones.

In the following of this Section, we discuss (without the ambition of being exhaustive) the approaches in the above two classes that most closely relate to our work.

A. Middleware-Based Approaches

A variety of middleware approaches, serving different specific purposes and for different application areas, recognize the suitability of interaction models based on local shared data spaces [7], [21], [31]. A representative example is the Gaia active spaces infrastructure for pervasive collaboration environments [31]. The common basic idea, shared by our proposal, is that shared data spaces can enable dynamic location-dependent interactions between software components and adaptive fruition of services in mobile settings. These environments recognize, as our proposal does, the need to clearly separate the intracomponent aspects from the inter-component ones, the latter being delegated to configuration tools in the shared data spaces. However, they usually define limited configuration capabilities, mostly based on simple declarative approaches or on simple scripting (as in Gaia). Thus they lack the fully programmable power that we ascribe to local service contexts and that is implemented in MARS tuple spaces.

Other middleware systems based on shared data spaces and, more generically, on distributed data structures, have been proposed with the goal of supporting and facilitating contextual and location-dependent coordination activities in the presence of mobility. Although explicitly conceived to support coordination services, these systems can be conveniently exploited also to support M-services. For instance, the Lime [29] and the XMIDDLE [26] middleware systems propose relying on shared data structures (tuple spaces and XML trees, respectively) as the basis for both supporting coordination and context-awareness. Each mobile device/agent in the network owns a private data structure (e.g., a private tuple space or a portion of an XML tree). Upon connection with other devices/agents (either in an ad-hoc network or within a host in the fixed network infrastructure), the privately owned data structures can merge together accordingly to specific policies. The resulting merged shared data structure intrinsically provides a location-dependent perspective, and can be used by agents as a common interaction space to exchange contextual information, to coordinate with each other, and to access services. However, these models offer only a limited form of user-level and administrator-level programmability, lacking the full programmability of our service contexts and missing in promoting a clean separation of concerns in application design and development.

A variety of systems, without explicitly focusing on coordination and adaptation in accessing services, aim at defining tools to facilitate the dynamic, location-dependent, discovery of services. The approach proposed in [30] – and also supported by the TOTA middleware [25] – enable each mobile node in a network to specify an interest for some location-dependent information, together with the scope of that interest (e.g., all the gas station within ten miles). The middleware is then in charge to build a distributed data structure (i.e., a shortest path tree) spanning all distributed nodes within the scope of the interest. This data structure will then be used to route back to the interested 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 constraints and without requiring any centralized service (all nodes cooperate to provide the information in a distributed way). However, the proposal lacks flexibility, in that is purely focused on information gathering, and pays little or no attention to the problem of coordinating the activities of application components within a locality. The approach described in [12] proposes a middleware architecture for dynamically binding a mobile client to local services. The user specifies, via a high-level declarative language, the characteristics of the required services. The middleware, upon movements of the client, looks for the locally available services those that best match the clients' preferences. We are aware that our proposal is currently a bit limited with this regard: an agent on a mobile device can typically access only the service on the local service context (i.e., the local MARS tuple space) to which it is currently locally bounded. More flexibility, by enabling coordination among service contexts and access to a multiplicity of distributed service contexts may be required.

B. Agent-Based Approaches

Agent-based approaches support adaptive M-services by exploiting the capabilities of autonomous software agents, moving across a network [32] and interacting and negotiating with each other [19]. Of course, any agent-based approach requires the support of an agent middleware to support agent execution and communication. The point is that, in agent-based approaches, the agents and not the middleware are in charge of taking care and enabling an adaptive, context-aware, access to services.

The LEAP system [4], a FIPA compliant agent system [3], supports the ubiquitous provisioning of services to mobile devices (i.e., to mobile users) via high-level interactions between agents. A mobile device should locally run a nomadic LEAP agent, to act both as personal user assistant (in charge of interfacing with the user) and as repository of local contextual information (e.g., local bandwidth and device characteristics). Such agent is also in charge of interfacing, via high-level communication languages (i.e., FIPA ACL), with service LEAP agents on the fixed network infrastructure [3], [10]. Service agents are stationary ones, and typically co-located with those services that are to be made available to mobile users. They are in charge of wrapping a service and of acting as mediators between the service and the nomadic agents. The service agent and the nomadic agent negotiate to assess the way a service should be provided, on the basis of the local contextual information available to the nomadic agents. For instance, the nomadic agent can detect a very low available bandwidth and adisplay of limited capabilities, and may negotiate with the service agent a provisioning of the service suitable to those conditions.

An approach very similar to that of LEAP is the one promoted by the SOMA platform for mobile services [2]. There, the nomadic agents executing in the portable devices of LEAP are substituted by mobile agents associated to each user and executing on the fixed network infrastructure. A mobile agent is capable of following a user in its movements, typically by executing on to (or close to) the access points the user is currently exploiting to access the fixed network infrastructure. As in LEAP, the mobile agent on the fixed infrastructure is capable of detecting the local conditions and adapts the fruition of the service to specific location-dependent conditions.

Both the above approaches effectively provide a solution to the need of adaptable accessing to M-services. Also, by moving at the level of programmable software agents the problems related to such adaptation, they overcome the limitations in flexibility of those middleware-based approaches hardwiring the adaptation logics. The problem is that the adaptation logics, being associated to specific software agents, can hardly take into account location-dependent and group-dependent issues, as in our proposal. Moreover, our proposal (despite being mostly a middleware-based one) does not exclude and instead promotes the use of specialized software agents to mediate the access to local service contexts. Although we have not explicitly dealt with such an issue in this paper, it is clear that such agents could take care of adapting M-services fruition to the specific needs of a specific agent/user. So, it is definitely possible to couple the agent-based facilities proposed of systems like LEAP and SOMA with the possibility of accessing to a world of location-dependent and group-dependent local service contexts.

Another interesting approach that somewhat provides a good compromise between a middleware-based and an agent-based approach is presented in [24]. There, the fruition of mobile services is supported by a set of software agents that interact with each other in the context of a shared virtual space called "meeting infrastructure." In the meeting infrastructure, agents representing both M-service providers and users can meet and interact directly with each other to negotiate the provisioning of a service. The meeting infrastructure is in charge of monitoring the interactions that occur within, of enforcing security in service provisioning, and of directing user agents toward the most appropriate service agents. Despite quite close in spirit, the meeting infrastructure and our proposal exhibit key differences. First, being accessible only from a locality, local service context enforce location-awareness and location-dependency, while the meeting infrastructure fully disregard this issue. Secondly, the activities of the meeting infrastructure are fixed once and for all and cannot be programmed by users and/or administrators. Nevertheless, as already stated, the idea of coupling a coordination infrastructure with the smart capabilities of software agents is an effective one, and it can be easily supported in our proposal.

VI. Conclusions and Future Work

The effective fruition of M-services by mobile users and, more generally, by mobile agents, should enable flexible adaptation to the current user location. In this paper, we have introduced a framework that promotes a modular and flexible approach to the design of distributed applications exploiting location-dependent M-services. Centered around the concept of active and programmable local service contexts, the framework enables the fruition of M-services (whether related to the access of some resources or inter-agent coordination services) to be dynamically adapted to the current location of a user/agent, whether such location is physical or simply logical. Moreover, we have shown how the MARS programmable coordination infrastructure, by mapping at the infrastructure level the abstractions of the framework, can lead to the development of easy to program and easy to maintain mobile applications.

At the time of writing, we are working along several directions to improve both our framework and the MARS infrastructure. First, we are trying to extend the model so as to consider explicit coordination among local service contexts, in order to somehow extend the strictly local and sometimes limiting perception of services by agents. Second, we are evaluating how and to which extent the presented model could be extended for applications in the area of mobile ad-hoc networks [6], to promote the adaptive fruition of services even in the absence of a fixed network infrastructure. Third, we are developing a MARS implementation that takes into account the emerging standards for M-Services. At the moment, MARS is compliant with Sun's JavaSpaces and can be accessed as a Jini service. We further plan letting the tuple space being accessible through Simple Object Access Protocol (SOAP) calls, to encode tuples in Web Service Description Language (WSDL), and to implement a matching process based on the Universal Description, Discovery and Integration protocol (UDDI).

REFERENCES

[1] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *IEEE Computer*, vol. 19, pp. 26–34, Aug. 1986.

[2] P. Bellavista, A. Corradi, and C. Stefanelli, "Mobile agent middleware for mobile computing," *IEEE Computer*, vol. 34, pp. 73–81, Mar. 2001.

[3] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi-agent systems with a FIPA-compliant agent framework," *Software—Practice and Experience*, vol. 31, no. 1, pp. 103–128, Jan. 2001.

[4] F. Bergenti and A. Poggi, "Ubiquitous information agents," *Int. J. Cooperat. Inform. Syst.*, vol. 11, no. 3, pp. 231–244, 2002.

[5] M. E. Bonfigli, G. Cabri, L. Leonardi, and F. Zambonelli, "Virtual visits to culturale heritage supported by web agents," J. Inform. Softw. Technol., to be published.

[6] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva, "A perfomance comparison of multi-hop wireless Ad Hoc network routing protocols," in ACM/IEEE Conf. Mobile Computing Networking, Dallas, TX, Oct. 1998.

[7] G. Cabri, L. Leonardi, and F. Zambonelli, "Mobile-Agent coordination models for internet applications," *IEEE Computer*, vol. 33, pp. 82–89, Feb. 2000.

[8] ——, "MARS: A Programmable coordination architecture for mobile agents," *IEEE Internet Comput.*, vol. 4, pp. 26–35, July-Aug. 2000.

[9] ——, "Engineering mobile agent applications via context-dependent coordination," *IEEE Trans. Software Eng.*, vol. 28, pp. 1040–1056, Nov. 2002.

[10] G. Caire, N. Luhillier, and G. Rimassa, "Communication protocols for agents on handheld devices," in 1st Int. Workshop Software Agents Pervasive Ubiquitous Environment at AAMAS, Bologna , Spain, July 2002.

[11] A. Campbell, J. Gomez, S. Kim, A. Valko, and C. Wan, "Design, implementation, and evaluation of cellular IP," *IEEE Pers. Commun.*, vol. 7, pp. 42–49, Aug. 2000.

[12] A. Cole, S. Duri, J. Munson, J. Murdock, and D. Wood, "Adaptive service binding to support mobility," in *23rd Int. Conf. Distributed Computing Systems Workshops*, New Providence, RI, May 2003, pp. 369–374.

[13] G. Cugola, A. Fuggetta, and E. De Nitto, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 827–850, Sept. 2001.

[14] E. Denti, A. Natali, and A. Omicini, "On the expressive power of a language for programmable coordination media," in Proc. 10th ACM Symp. Applied Computing, Atlanta, GA, Mar. 1998.

[15] A. K. Dey and G. Abowd, "The context-toolkit: Aiding the development of context-aware applications," in *Proc. Conf. Human Factors in Computing Systems*, New York, 1999, pp. 434–441.

[16] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces: Principles, Patterns, and Practice*. Reading, MA: Addison-Wesley, 1999.

[17] A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility," *IEEE Trans. Softw. Eng.*, vol. 24, pp. 352–361, May 1998.

[18] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 96–107, Feb. 1992.

[19] N. R. Jennings, "An agent-based approach for building complex software system," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, Apr. 2001.

[20] T. G. Kanter, "Attaching context-aware services to moving locations," *IEEE Internet Comput.*, vol. 7, pp. 43–51, Mar.-Apr. 2003.

[21] T. Kindberg and A. Fox, "System software for ubiquitous computing," *IEEE Pervasive Computing*, vol. 1, pp. 70–81, Jan.-Mar. 2002.

[22] Z. Maamar, B. Benatallah, and Q. Sheng, "Toward a composition framework for E-/M-services," in 1st Int. Workshop Software Agents Pervasive aUbiquitous Environment at AAMAS, Bologna, Spain, July 2002.

[23] Z. Maamar, E. Dorion, and C. Daigle, "Toward virtual marketplaces for E-commerce," *Commun. ACM*, vol. 44, no. 12, pp. 35–38, Dec. 2001.

[24] Z. Maamar, W. Mansoor, and Q. Mahmoud, "Software agents to support mobile services," in *Proc. 1st Int. Joint Conference Autonomous Agents Multiagent Systems*, Bologna, Spain, July 2002, pp. 666–667.

[25] M. Mamei, F. Zambonelli, and L. Leonardi, "Tuples on the air: A middleware for context-aware computing in dynamic networks," in *Proc. 23rd Int. Conf. Distributed Computing Systems Workshops*, New Providence, RI, May 2003, pp. 342–347.

[26] C. Mascolo, L. Capra, and W. Emmerich, "XMIDDLE: A data-sharing middleware for mobile computing," *Pers. Wireless Commun. J.*, vol. 21, no. 1, pp. 77–103, Apr. 2002.

[27] N. H. Minsky and V. Ungureanu, "Law-Governed interaction: A coordination & control mechanism for heterogeneous distributed systems," *ACM Trans. Softw. Eng. Methodology*, vol. 9, no. 3, pp. 273–305, July 2000.

[28] C. E. Perkins, *Mobile IP: Design Principles and Practice*. Reading, MA: Addison-Wesley, 1997.

[29] G. P. Picco, A. L. Murphy, and G. C. Roman, "LIME: A Middleware for logical and physical mobility," in 21st Int. Conf. Distributed Computing Systems, July 2001.

[30] G. C. Roman, C. Julien, and Q. Huang, "Network abstractions for context-aware mobile computing," in *Proc. 24th Int. Conf. Software Engineering*, Orlando, FL, May 2002, pp. 363–373.

[31] M. Roman, B. Ziebart, and R. H. Campbell, "Dynamic application composition: Customizing the behavior of an active space," in *Proc. 1st IEEE Conf. Pervasive Computing Communications*, Dallas, TX, Mar. 2003, pp. 169–176.

[32] J. White, "Mobile agents," in *Software Agents*. Menlo Park, CA: AAAI Press, 1997, pp. 437–472.

[33] M. Wooldridge, N. R. Jennings, and D. Kinny, "The gaia methodology for agent-oriented analysis and design," *J. Autonom. Agents Multi-Agent Systems*, vol. 3, no. 3, pp. 285–312, 2000.

[34] F. Zambonelli *et al.*, "Agent-Oriented software engineering for internet applications," in Coordination of Internet Agents, A. Omicini *et al.*, Eds. New York: Springer-Verlag, 2001.

[35] F. Zambonelli, N. R. Jennings, and M. J. Wooldridge, "Organizational abstractions for the analysis and design of multi-agent systems," in *Agent-Oriented Software Engineering*. New York: Springer-Verlag, 2000, vol. 1947, LNCS.

Giacomo Cabri received the Laurea degree in electronic engineering from the University of Bologna, Bologna, Italy in 1995, and the Ph.D. degree in computer science from the University of Modena and Reggio Emilia, Modena, Italy in 2000.

He is a Research Associate in computer science at the University of Modena and Reggio Emilia, since 2001. His research interests include methodologies, tools and environments for agents and mobile computing, wide-scale network applications, and object-oriented programming.

Letizia Leonardi received the Ph.D. degree in computer science from the University of Bologna, Bologna, Italy in 1989.

She is a Full Professor in the Department of Information Engineering at the University of Modena and Reggio Emilia, Modena, Italy since 2001. Her research interests include design/implementation of mobile-agent systems.

Dr. Leonardi is a member of TABOO and AICA.

Marco Mamei received the Laurea degree in computer science in 2001 from the University of Modena and Reggio Emilia, Modena, Italy where he is currently pursuing the Ph.D. degree in computer science.

His current research interests include distributed and pervasive computing, complex and adaptive systems, multiagent systems.

Mr. Mamei is a member of AIIA and TABOO.

Franco Zambonelli (M'97) received the Laurea degree in electronic engineering in 1992, and the Ph.D. in computer science in 1997, both from the University of Bologna, Bologna, Italy.

He is Professor of computer science at the University of Modena and Reggio Emilia, Modena, Italy since 2001. His current research interests include distributed and pervasive computing, multiagent systems, agent-oriented software engineering.

Prof. Zambonelli is member of the Management Committee of the European Network of Excellence "Agentlink II" and, within the same network, Coordinator of the Special Interest group on "Methodologies and Software Engineering for Agent Systems."He is a member of ACM, AIIA, and TABOO.