The UbiMedic Framework to Support Medical Emergencies by Ubiquitous Computing

Francesco De Mola [1,2] Giacomo Cabri [1] Nicola Muratori [3] Raffaele Quitadamo [1] Franco Zambonelli [2]

[1] Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia, 41100 Modena, Italy
Email: {cabri.giacomo; demola.francesco; quitadamo.raffaele}@unimore.it
[2] Dipartimento di Scienze e Metodi dell'Ingegneria – Università di Modena e Reggio Emilia, 42100 Reggio Emilia, Italy
Email: zambonelli.franco@unimore.it
[3] O2 Studio Ingeneri Associati di Muratori Nicola e Rossi Massimiliano, 42020 Albinea, Reggio Emilia, Italy
Email: nicola.muratori@oduestudio.it

Abstract: This paper investigates the feasibility of employing the Software Agent technology in the highly dynamic and variable context of healthcare emergency coordination and decision-support domain. We introduce the design of an agent-based middleware tailored to the requirements of such context and propose a framework, called *UbiMedic*, for the implementation and deployment of services, like monitoring services, communications and remote medical measurements in injured people. From the analysis of the framework, we are able to identify some of the major technical requirements it should meet as well as challenges to be addressed for effective use in commercial applications. We choose software agents as the key enabling technology because they offer a single, general framework in which large-scale, distributed real-time decision-support applications can be implemented more efficiently.
Keywords: Pervasive computing, multi-agent systems, context awareness, medical applications.

1. Introduction

Software agent technology provides an attractive and important model for building large-scale distributed applications in heterogeneous computing environments. In particular, a "mobile" agent can be viewed as an autonomous program that has the ability to transport itself between the nodes of a network entirely under its own control, carrying with it the data and the execution state required to resume execution at the destination host [5]. Due to their autonomous and active nature, agents offer several benefits over traditional technologies such as the client-server paradigm, to enable a mechanism for context-aware decision support. Being able to encapsulate and transfer the know-how needed to perform a given task, agents tend to be relatively easy to customize and can rapidly adapt to changing user requirements and run-time context. Some of current pervasive computing research projects combine agenthood with context awareness [10] and, through this convergence, mobile agents are capable of embedding light-weight context reasoning engine and can do preliminary processing in the original context.

The healthcare field is not only widely distributed and fragmented but it also exhibits a high degree of heterogeneity. The current lack of standards across different institutions and within the same institution prevents it from using a single software solution to support a cooperative working environment. The framework presented in this paper focuses primarily on the domain of medical emergency management, which is a very information-intensive and mission-critical one. In such scenarios, the hospital environment by default is 'highly mobile' with caregivers constantly on the move. In order to meet the varying information and resource needs of these personnel and yet be able to support their physical mobility requirements, agent-oriented computing seems to provide an ideal fit. The architecture designed in this paper draws inspiration from the medical domain, representing an emblematic application scenario that well includes all the seen features. However, it lays the foundations of a system exploitable in a quite wide range of contexts, where context-awareness and heterogeneity are to be faced in a distributed scenario. The agent paradigm offers a great flexibility, but it comes also with some major challenges that are infrequent in the client-server paradigm (e.g., the key issues of security and trust) and that we try to clearly highlight here.

In this paper, we advocate the pro-active, goal driven and autonomous nature of software agent technology as a support to cope with the highly mobile, dynamic and variable context of medical emergencies. Starting from this consideration, we propose an agent-based framework, named *UbiMedic*, as a helpful abstraction level upon the heterogeneity of medical emergency scenarios: our basic idea is that every entity involved in a healthcare environment (e.g., medical instruments, ambulances, doctors) should be represented by an agent installed in the platform. Consequently, application specific services (e.g., remotely driving medical devices, chatting with doctors, coordinating the overall emergency operations) are carried out instantiating proper agents and letting them interact, in a ubiquitous and context-aware fashion.

The paper opens up showing the main requirements for emergency (Section 2) and a short report on the state of the art (Section 3). After some methodological considerations about agents (Section 4), a global presentation of UbiMedic is provided in Section 5. A detailed analysis of the middleware components (Section 7) is followed by a deeper description of the adopted solutions for the context management (Section 8). Finally, the paper ends with critical

evaluation and future challenges (Section 9), just before conclusions (Section 10).

2. Emergency Scenario Requirements

The field of medical emergencies is much more dynamic and subject to context variability, compared to other telemedicine fields. It often presents difficulties in communications and even in physically reaching the place where first aid is needed. The efficiency of the assistance is an aspect of vital importance, so that all involved people (from healthcare workers to drivers) need to communicate, coordinate and access distributed resources in a very simple and fast way. Moreover, the scene of the accident is not a priori known and the employed resources (users and medical devices) must dynamically organize themselves in a temporary network, where communication links are established just in time. The technology to set up the communication links must also be chosen in real time depending on the particular context.

Therefore, it clearly emerges that the design of a suitable framework should fulfill some major requirements: first and foremost, it should provide an adequate degree of *dynamicity* and *context-awareness*, i.e. services should easily self-configure and adapt to the situation variability. The system must support the emergency staff to remotely monitor and coordinate the distributed resources in real time, allowing to successfully control critical situations and patients' health. In order to make the users as accustomed and familiar as possible to the system, *portability* is another important requirement to take into account, so that each operator can seamlessly access the system from her preferred machine. In addition, the integration of different devices and applications (*interoperability*) through open-source technologies and common standards brings further benefits: thanks to integration, users can access different services from the same device; distinct systems can communicate and interact more smoothly; the existing equipment of the healthcare service can interoperate with new systems and devices without headaches due to proprietary solutions and incompatibilities; efforts and results coming from different experiences can be unified and integrated to obtain a more and more complete system. Integration must be intended also in communications, combining multimedia means and different channels: on the one hand, users must have the possibility to choose the best communication mode (textual, vocal, video or combined) according to their needs; on the other hand, the system must automatically choose the most efficient communication technology among the available ones at runtime. *Reliability* and *fault-tolerance* to human errors are, in fact, an indispensable requirement in such a delicate field as emergency healthcare.

3. State of the Art

The contemporary scenario of telemedicine applications presents several technological solutions, spread over different fields and use cases. However, most of the applications actually used in health care are often strictly confined to each particular experience, referred to specific companies and localities.

For example, let us consider the *LIFENET® System* by Medtronic Physio-Control [8], a multinational company that provides a full range of services and complementary products that form an emergency cardiac care system. Twelve-lead electrocardiograms and other vital signs are measured on the patient through a device called *Lifepak 12*, a portable multifunctional monitor and defibrillator. This information can be transmitted by a Bluetooth connection toward the doctor's mobile phone or a *Lifenet EMS* (Emergency Medical System). This device, belonging to the ambulance equipment, is an electronic patient care reporting system (ePCR) implemented on a tablet PC: it can collect, elaborate and show all the data received from the Lifepak. Moreover, the same data can be sent by GSM or GPRS from the mobile phone to a receiving station (*Lifenet RS*), a PC installed in the hospital, where a specialist can give a second opinion to the doctors directly involved in facing the emergency.

This solution, as well as the other commercial products, has the heavy limitation of being completely closed and protected by patent rights, so that the system cannot communicate and integrate with other solutions. The limited possibility to access several services through a single integrated system is often underlined by staff in charge of medical and territorial emergencies. This implies also efficiency problems in resource employment and integration, because devices previously in use in a hospital can hardly be integrated with a new system made by a different company.

Moreover, the information technology is prevalently applied in telemedicine among hospitals and fixed health care centers, with the implementation of tele/videoconference and data transmission systems to ask for second opinions and to follow patients and treatments at a distance. Instead, the external scenario of medical and territorial emergencies still need deeper exploration, having to cope with a much more variable and complex context.

We can find interesting solutions in the field of eldercare [1], such as the six-month study conduced at the Honeywell Laboratories to implement the *Independent LifeStyle Assistant* (ILSA) [6]. This is an agent-based monitoring and supporting system to help elderly people to live more independently at home, by reducing caregivers load. It consists of a multi-agent system supporting continuous data monitoring via home-installed sensors. The collected data are processed to obtain response planning and machine learning. In particular, ILSA is implemented over the JADE agent platform. The project development met serious difficulties due to the use of agents: the agent paradigm turned out to be too expensive for a system presenting many centralized features, so that a simpler design would have been sufficient and more convenient.

On the other hand, if we consider the CodeBlue [13] experience, it seems closer to the emergency field, where distributed features are much more marked: CodeBlue is a combined hardware and software platform for medical sensor networks developed in a prototypal version at the Harvard University. Sensor networks consist of small, low-power and low-cost devices with limited computational and wireless communication capabilities. CodeBlue comprises a suite of protocols and services that let many types of devices coordinate their activities. Once again agents appeared unsuitable because of the extremely limited resources of the devices, where also other traditional approaches (like RPC and JVM) are not feasible for the same reason. Furthermore, the medical devices used to test CodeBlue, like a pulse

oximeter and a two-lead electrocardiogram monitor, were developed ad hoc for the system.

4. The Agent Methodology

Many peculiar issues in the medical domain, in particular those belonging to the emergency field, can be addressed very well with the conceptual and methodological tools provided by the agent technology [9]. Several reasons can motivate such a choice:

- each component of a multi-agent system can run on a different machine, fitting very well distributed scenarios, such as the emergency assistance one;
- thanks to their *sociality*, agents can interact, coordinate and cooperate to reach common objectives, reflecting what the different emergency units must actually do in their work;
- medical domain problems are intrinsically complex: multi-agent systems adopt decomposition techniques to tackle complex problems, partitioning the problem space into smaller tasks; this helps the system's designer focusing on some simpler portions of the system, deferring global more complex decisions to the last design stages [2];
- *proactivity* is another feature of software agents: a proactive agent is able to select useful actions depending on the perceived context and the explicit user intervention; this can facilitate the course of operations when maximum efficiency is required;
- the agent paradigm provides a model in which autonomous entities (i.e. agents) are assigned high-level goals to achieve and they have the reasoning capability required to take decisions by their own: they interact and mutually cooperate, keeping at the same time a certain degree of independence related to internal organization, information ownership and responsibility level, fitting very well the different institutions involved during an emergency (e.g., hospital, fire brigade, civil defence,…);
- finally, agent mobility could be beneficial in external dynamic contexts, such as those of emergencies and accidents, where the location is an unpredictable variable.

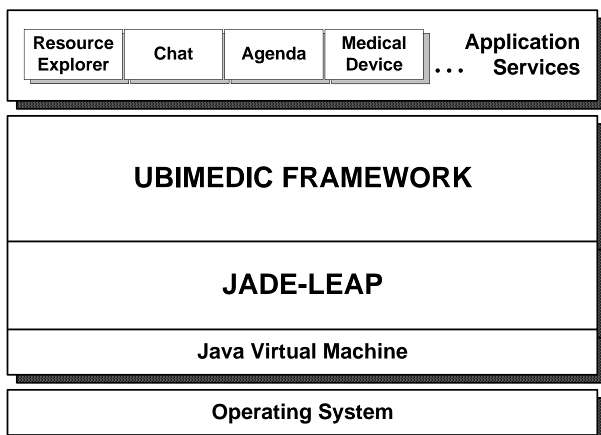

Fig. 1. A layered view of the architecture.

The adoption of the multi-agent paradigm, however, comes at the cost of an increased development effort. On the one hand, the agent paradigm provides a methodology to deal with the domain complexity using high-level abstractions, but on the other hand, the agent adoption itself can make the development process more expensive. Other experiences have already highlighted these difficulties [6], but we argue that an appropriate architecture is mandatory to tackle medical emergency issues. Taking into account such pros and cons of agent-based software engineering, we propose a useful middleware that provides a whole set of facilities for users and application services: the main prerogative of this system is that it can integrate even non agent-oriented services and applications, transparently enriching them with an "agent wrapper": through a properly studied architecture (described in the following Sections), the system can exploit the benefits of a MAS (Multi-Agent System) and limit the mentioned development process difficulties.

5. UbiMedic: A Prototype to Support Medical Emergencies

UbiMedic is a research work aimed at building a distributed context-aware platform with mobile agents implementing helpful services for user applications. The middleware is an agent-based framework and is designed taking into consideration the importance of portability even on small devices (e.g., PDAs or other limited user's terminals). Before starting the analysis of UbiMedic internals, some concepts are worth being clarified. UbiMedic is an extensible medical emergency platform, where many different *application services* can be deployed. These services comprise *user application services* and *built-in services*. The former are implemented by programmers as Java modules, easily installed into the system by the end users and can implement custom applications (e.g., special monitoring applications, a shared agenda, a chat, etc.). Built-in services are embedded into the platform and are usually general-purpose facilities offered to users and to other application services. The most interesting typology of built-in service is represented by *medical device drivers*. These are services used to access a medical device (e.g., an electrocardiograph or a pulse oximetry sensor) and are implemented by the various device manufacturers to facilitate the monitoring and control of their instruments.

The proposed system relies on a middleware built upon the operating system, whose services are available to several kinds of applications of the upper level (see Fig. 1). The middleware layers offer both low and high level services (discussed later), to grant dynamicity and context-awareness features to the system. In the following Subsections, we will examine one by one every layer of the UbiMedic architecture, starting from the application services layer.

6. Application Service Layer

The possible applications that can benefit from the facilities in the UbiMedic framework spread over a wide range of medical emergencies support tools: from a resource explorer, to textual and vocal chat, medical device interactions, on-map location, shared agenda, data storage, logging, administration tools, and so on. *Application services* are designed and implemented separately from each others: they can be installed as independent modules, progressively enriching the

system with new available functionalities. Furthermore, it is by no means required that these software modules are agent-based applications (e.g., they can be legacy Java applications). The seamless integration of non agent-oriented applications within an agent platform demanded the conception of a proper service model.

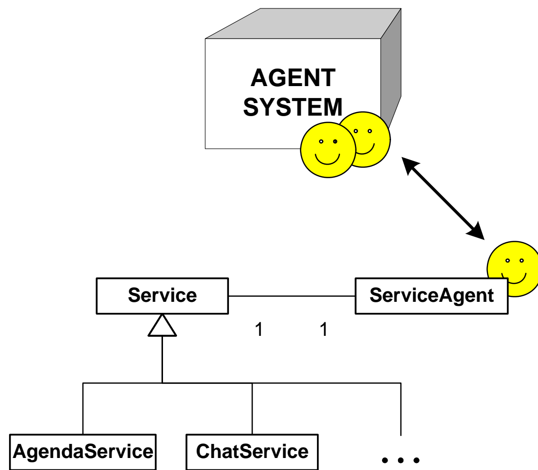

Fig. 2. The main classes of the applications.

An *application service* is composed of three entities (Fig. 2):

(i) the *Service* class, which exposes a public shared interface for a generic UbiMedic application, like the ones listed above;

(ii) an *application specific class*, extending the Service superclass, which contains the code implementing the particular functionalities of the application;

(iii) a *Service Agent*, univocally associated to each instantiated service, which is the entity representing the instance of the application in the agent platform; this agent is responsible for invoking the service methods of the application, acting as a mediator between the application classes and the rest of the multi-agent system.

In practice, if a developer wants to add a new application service, he has just to implement a class inheriting from *Service*, specifying its own logic. As mentioned above, the methods to interface the application class with the Service Agent are exposed as a standard interface by the Service class. As a consequence, each application is implemented as a plain Java application, ideally without any knowledge of the agent paradigm underneath. The advantage of this approach is in the fact that all those tricky issues, related to the used agent technology, are completely transparent to the user.

For example, we can list here a few essential application services:

Resource Explorer: each user connecting to the system will always start this service first. The application provides a complete graphical view of all the connected resources (users and devices). The view is personal, according to user's profile and visibility permissions, and can be organized in trees, grouping the identities on the base of their typology. The Resource Explorer, through the mediation of the Request Manager (see the next section), must interrogate principally the Discovery service (see later) to collect information about connected resources.

Chat: a real-time communication channel among connected users. It can be textual, vocal or video, to meet all coordination needs during an emergency. It can be started selecting the receiver from the list shown by the Resource Explorer.

The advantages of using an agent-based framework could be clarified if we refer to the potentialities added to applications thanks to the cooperative and proactivity properties of agents. In fact, let us consider, for example, a patient needing an emergency hospital admission: exploiting the agent social ability, the system can collect information about hospitals and means of transportation in that region and choose the best destination hospital according to several deciding elements (such as geographic distance, room and doctors availability, instrument and medicine equipment). For instance, the application could discover a helicopter approaching the region, so that other farther hospitals could be automatically taken into consideration for the best solution.

6.1.1 Medical Device Drivers: a Key UbiMedic Service

As already said, the interaction with a medical device is a first-class built-in application service in UbiMedic. The remote interaction with a medical device consists in visualizing its measured values and, if possible, in sending some requests to drive the device. Medical device drivers are agent-based services implemented by manufactures according to the general service model explained above. When a user (e.g., a specialist doctor) wants to verify the vital signs coming from a remote device, he has to ask the system for the *medical device service* suitable for that specific device. The distributed and unpredictable nature of remote device monitoring makes convenient having a more distributed implementation of the service than the centralized one. In other words, we still have a generic Service Agent class installed in UbiMedic, implementing many functionalities useful for the application services (e.g., communication and data logging facilities). Nonetheless, other important pieces of the service logic are scattered across a set of three kinds of agents instantiated for each device (see Fig. 3):

(i) a *User Interface Agent* (UIA) responsible for managing user interactions, by means of a more or less complex GUI (Graphical User Interface);

(ii) a *Physical Resource Interface Agent* (PRIA), which has to collect and make available, to requesters all over the platform, the patient's vital signs measured by medical devices;

(iii) a *Proxy Agent* (PA) which includes the proper logic to process and filter the collected data according to the specific source/device and to the particular goals of the application. This agent extends the Service Agent exploiting its general purpose facilities for application services.

Each time a user, through her graphical interface, wants to interact with a medical device, a proxy agent is generated and acts as a mediator between the two entities (i.e. the UIA and the PRIA). Having a PA associated with the UIA relieves the user’s device (maybe with limited resources) of the burden of executing the bulk of the code, letting the system choose the best location where to move and execute the PA, considering the amount and type of needed resources.

For example, let us consider a first aid case study, where an ambulance with some volunteers reaches the place of a car accident: the caregivers start the detection of a patient's

electrocardiogram, which is automatically transmitted to obtain some specialists' second opinion. The agent-based system is able to choose the most efficient available communication link, select all the actors that should be involved to obtain a second opinion (e.g., the connected cardiologist) and perform the data processing needed to send them the electrocardiogram. In fact, a doctor must be reachable even if he is moving with a limited device, like a cellphone, needing a proper transcoding of the information the device can't support by itself: a mobile PA comes in help performing the data transcoding on a more capable node. This way, we can achieve increased portability of user's applications on a wider range of computational devices.

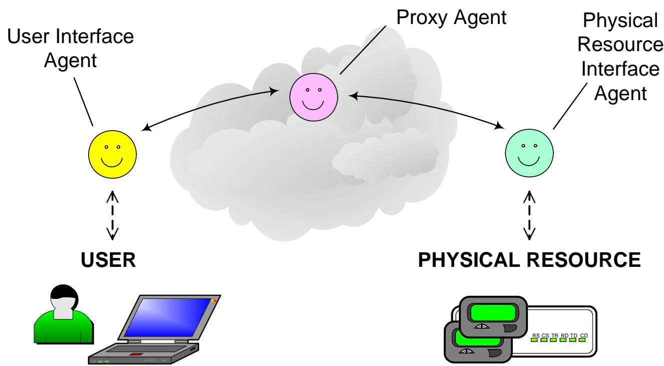

Fig. 3. The three agent typologies of a medical device application service

6.2 The Middleware

The middleware is composed of three main layers, described in the following. We chose to exploit existent and standard technologies to reduce the implementation work, to grant more flexibility and openness.
Just a layer upon the operating system, we have the *Java Virtual Machine (JVM)* as the common execution environment to deploy the same application on different systems and platforms. It is the lowest layer of the middleware and it helps achieving isolation from the underlying hardware/OS. There are different JVMs that can be installed and configured to run on resource-constrained devices (such as the Java 2 Micro Edition), granting a good degree of portability to the system.

A *multi-agent platform* provides all the facilities to administrate a multi-agent system, ensuring the needed supports for agent's life-cycle, their interactions, coordination and mobility. As already said, the agent paradigm can be the best fit to model a complex system in a dynamic and distributed environment, like healthcare and medical emergency scenarios. The chosen platform, JADE [7], is a FIPA-compliant framework, Java-based, providing some useful services to implement inter-agent communication, node UDP monitoring, security and fault-tolerance features. The LEAP distribution enables FIPA agents to execute also on lightweight devices, such as mobile phones or PDAs running the JVM.

The proposed *UbiMedic framework* is a layer of the middleware implementing context-awareness over the JADE multi-agent platform. It is composed of several modules, grouped into two main levels: *low level services,* which are responsible for environment monitoring and the interactions with it; *high level services,* implementing more sophisticated and evolved functionalities, directly used by the application layer. The UbiMedic framework is detailed in the next section.

7. UbiMedic Framework

UbiMedic is an agent-based middleware, implemented on top of JADE. Fig. 4 shows the low and high level services composing the framework, detailed in the following subsections. We anticipate that each module in the presented architecture is implemented by means of one or more agents (e.g., the Authorization Agent or the Session Factory Agent). The scope of these agents will be clearer in Section 6.

Before diving into UbiMedic internal services, a few words must be spent to explain our vision of the "context" concept, because this will make the subsequent discussion easier to understand. We found useful to distinguish between *session context* and *physical environment context*: the former refers to the situation of each identity in the system, its preferences, authorizations and policies; the latter comprises instead information about the surrounding environment, i.e. device computational capabilities, connectivity and so forth. Every change in both the context typologies triggers an *event* that must be collected and efficiently delivered to the interested agents in the system: this is exactly the functionality of the vertical module *Event Manager* depicted in Fig. 4 and discussed later.

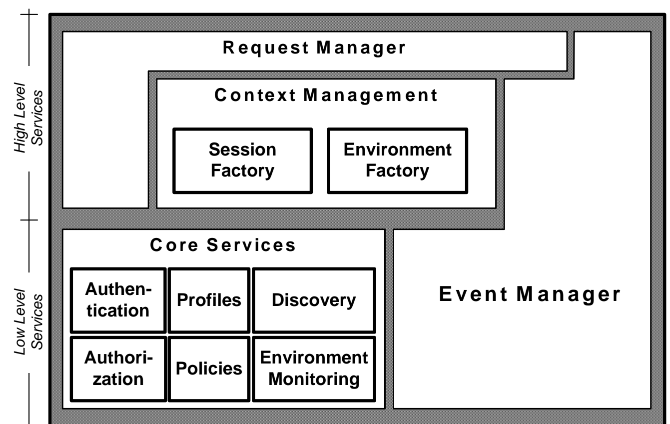

Fig. 4. The UbiMedic framework

7.1 Low Level Services

Low level facilities include six modules representing the core services of the system:

Authentication. This module verifies the credentials of the identities trying to access the system. By "identity" we mean not only each password-enabled user, but also all the active resources (e.g., medical devices) which connect to the system. Since a user can access the system through different devices, he must each time manually specify his credentials; the medical device credentials are automatically exposed instead by the corresponding Physical Resource Interface Agent (PRIA), once properly preconfigured. The Authentication module allows new connection requests if the requesting identity has the right credentials, which are maintained in a database, associated to each identity.

Authorization. It is the facility governing all the actions that identities can perform after joining the platform. Permissions and rules are stored in a database according to the RBAC model [12], where multiple roles can be assigned to each identity, and each role can be assigned multiple permissions. The defined permissions regulate each possible action, from resource access to agent social life. No action can be performed without the acknowledgement of the Authorization module. More details about the RBAC implementation are discussed in Section 8.3.

Profiles. We call "profiles" all the generic parameters defining some identities' preferences, such as user's customization and device working parameters. The corresponding core service, wrapped in the "profiles" module, retrieves these fields once again from a database and updates them, when any parameters of a preference are modified.

Policies with the term "policy" we intend preconfigured operations an identity must perform in response to some trigger events happened in the environment. This module allows the association of these actions to each identity in the platform, in a similar way to permissions and profiles.

Discovery. This module monitors in real time the available resources and the identities present in the system. Its aim is to show a continuously updated vision of the environment where the identities live. It acts as a mediator between the connected identities, which have to register in and deregister from the discovery module, and the below JADE directory and yellow pages services. The Discovery module is able to automatically perceive changes in the connected identities and to start searching agents according to the identity's requirements and the registered features.

Environment monitoring. This is a monitoring service of all the context features in the physical environments where each identity is running. The physical environment can dynamically change depending on the localization of a medical device, its computational capabilities, the availability of network access point and the occurring of some external events. While the previous modules are mainly about context session information, the Environment Monitoring deals with the physical aspects of the context. Together they constitute the primary information-collecting phase, at the base of context-awareness implementation.

7.2 The Event Manager

This module collects all the context change notifications, received from the application level and the other middleware modules, and dispatches them to the interested entities The Event Manager, differently from the other modules, can be considered a vertical service because of its ability to interact with both low and high middleware layers and application services. To overcome the disadvantages related to the centralization of this service (such as bottleneck and a lot of traffic only for the notification transmission) the system can be configured as a federation of Event Managers, each one competent in a limited locality, achieving better system scalability (see Section 1.1 for further design issues on this topic).

7.3 High Level Services

High level facilities can be divided into *context management* and *request management*. The C*ontext Management* layer manages context information of each identity in the system and defines the data structures to keep this information updated. It is composed of two modules, reflecting the context partition previously introduced:

Session factory. This module generates the necessary objects to let identities access the system. As already said, by session information we mean all those roles, permissions, profiles and policies related to an identity. Whenever a user or a resource attempt a connection to the system, the Session Factory collects all session information provided by core services (querying in particular the Authentication, Authorization, Profiles and Policies agents) and assigns them to a particular agent, called *Context Agent* (see Section 8.1 for more details), univocally associated with the new session of the identity. The identity is represented hereafter in the system by its Context Agent, containing all the context information initially provided by the Session Factory and kept updated through the interaction with the Event Manager.

Environment factory. This module collects all the physical context information perceived by the low level Environment Monitoring service and generates an *Environment Context Agent* related to the particular location where an identity is running. This agent encapsulates context information that are location-dependent (each node has its own associated Environment Context Agent): if the identity moves to a different location, or the surrounding environment changes, the Environment Factory transparently performs the substitution of the old Environment Context Agent or it updates its information.

The *Request Manager* is instead the layer that receives and tries to satisfy the requests of the various identities to access resources and functionalities. The identities have always to contact the Request Manager before performing any action in the system. Each request is analyzed and executed according to the context state: the action is performed only if both the physical context (e.g., availability of memory and bandwidth) and the session information (e.g., permissions and profiles) are consistent with its execution. The Request Manager works in tight relationship with the Context Management layer to know when to allow the received request. If allowed, the requested resources or the execution results are returned to the initial identity.

8. Context Management Details

UbiMedic requires context-awareness for the correct execution of the applicative services. For example, the framework must perceive if the bandwidth falls down to timely switch the ECG transmissions in progress on a more available communication channel; in addition, the framework is expected to instantly discover and update the list of connected identities, to let practitioners always be aware of the human and physical available resources.

In the following subsections we focus on the UbiMedic context management, showing in deeper details the existing interactions among the involved agents and discussing some distributed solutions to improve scalability. In particular we will consider "authorization" as one of the aspect representing a significant part of context information.

8.1 Context Generation and Update

As we noticed above, the services at the application level and the identities that joined the platform (both human and physical resources) must be informed of context changes. The mentioned *Context Agent* is associated to each identity accessing the system. The Context Agent is the entity representing a person or a device in the agent platform: it includes and keeps up to date all the context information of the related identity, making them available to other applications and identities requiring this kind of context awareness. Let us consider a distributed application that has to transfer data to a remote device: this application must be aware of the context of the receiver device, e.g., it must be aware of the data format understood by the receiver and of its computational capabilities, in order to perform the necessary transcoding before delivering the data.

The Context Agent is the contact agent to begin an interaction with the person or the physical resource it is related to. It provides all the information about the considered identity gathered from the low level services locally running on the node where the identity lives and from the centralized services keeping its personal settings. When a new identity joins the platform and its Context Agent is created, the Environment Factory Agent and the Session Factory Agent collect the needed information: in particular, the former contacts the low level services providing information about the local physical environment (transition number 1 in Fig. 5), and the latter interacts with those services in charge of providing general parameters and permissions related to the identity (transition 3 in Fig. 5). Both the agents send the obtained information to the just created Context Agent (transitions 2, 4). Since that moment, every change of these information are notified to the Context Agent, which must always be up to date. To this purpose, we introduced the already quoted *Event Manager*: low level services always notify any detected modification in their information domain to the same Event Manager (transition number 5). The Event Manager will forward the notifications to the interested agents, like the Context Agents of the identities involved in the context modifications and/or the application services interacting with the same entities (transitions 6, 7). An a priori registration to the Event Manager's notification service is needed: each agent must contact the Event Manager to specify the kinds of events it is interested in. The registration can be done at instantiation time and successively updated during all the agent life cycle. Similarly to the low level services, also the applicative ones can notify context modifications (e.g., the kind of currently active application services for each identity): the context events are always sent to the Event Manager (transition 8) and then delivered to the proper receivers, both Context Agents and other application services (transitions 9, 10).

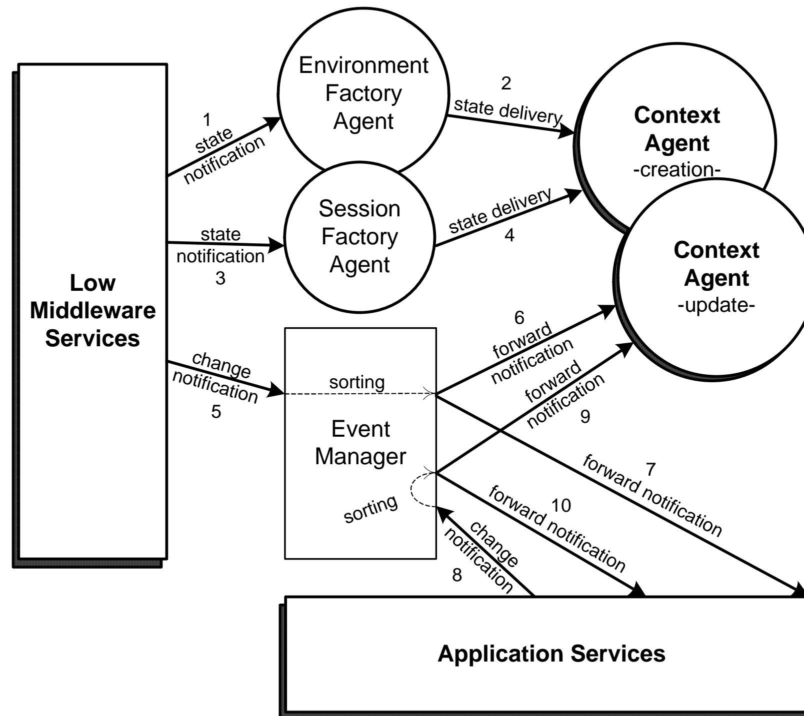

Fig. 5. General context management representation.

8.2 Event Management Issues

As just noticed, the Event Manager waits for subscription requests coming from all the agents interested in the context changes, receives all the context event messages in the whole platform and delivers them to the proper subscribed agents. If this *centralized solution* makes the event management easier to develop (for the logical separation of event detection from routing and message delivery), on the other hand it would likely be inefficient because of the huge amount of traffic generated by the events inevitably produced in dynamic contexts, like the emergency ones. Alternatively, a *fully distributed solution* would mean a relevant increase of the system complexity, for the intelligence each low middleware service should have to deliver their notifications in a sort of "network of scattered event managers". Besides, the identity

subscriptions for context change notifications should be replicated for each service and node, with additional resource consumption and consistency issues.

An *intermediate solution based on local areas* is proposed, combining the two approaches: the event management is decentralized in local areas, inside which a Local Event Manager has a centralized behavior analogue to the previously described one. Instead of a single Event Manager in the system, there can be several Local Event Managers, one for each identified local area (Fig. 6). First of all, the system must be divided in such local areas: they can be localized after analyzing the traffic generated from the context event notifications. The system administrator must then apply the best criteria (depending on the nature of the organizations adopting the system, with their own administrative rules and functional dependences among the actors) and try to optimize the area definitions, minimizing inter-areas communications. In other words, each area should group all the entities interested in the context changes regarding their own area. Each agent subscribes only to the Local Event Manager of its area and the generated notification traffic is mostly confined within such area. Whenever an identity needs notifications about the context of another local area (e.g., if an application service agent is communicating with an agent belonging to a different area, sending vital signs data for a second opinion by a remote doctor), it must still send the subscription request to his Local Event Manager, which will forward the request to the Local Event Manager of the interested area. When the second Event Manager detects a context change for which it received the subscription, it delivers the message directly to the interested identity in the first local area: this is the only inter-areas communication due to the context management the administrator must minimize with the most suitable area identification.

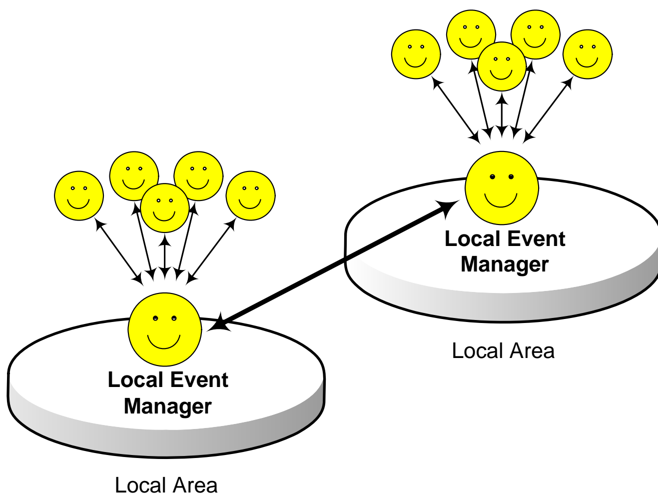

Fig. 6. Distributed solution for the Event Manager implementation.

A suitable implementation of this hybrid approach can be developed on the JADE framework, through the creation of multiple agent platforms, where each one fits a single local area. A single Local Event Manager lives in every platform and is responsible for keeping in touch with the adjacent Local Event Managers living in other platforms. New autonomous platforms can be increasingly added to UbiMedic, making the whole system more scalable.

Besides, this multiple platform solution fulfills also the autonomy requirement of the different organizations adopting the system, often very heterogeneous from each others. In fact, considering that different independent organizations can participate in the same emergency scenario (e.g., health care system and fire brigade), they could hardly accept to share the same platform regardless of their own internal rules and administration. In our agent perspective [2], each organization is free to define its internal business rules and agent responsibilities, including the preferred authorization system. These platforms can implement their own centralized services, without excluding the possibility of interaction between them, by means of inter-platform communication. Therefore, the coordination among the different organizations involved in the emergency care is allowed, though preserving their autonomy.

8.3 The Authorization Control

Authorization constitutes an important part of the contextual information to control the several actors in the platform. As pointed out in Section 7.1 an RBAC approach [12] is proposed, for the flexibility of adopting roles with permissions associated to the system identities. In the standard RBAC model there are two many-to-many relationships: between identities and roles and, on the other hand, between roles and permissions (Fig. 7 (a)). UbiMedic, instead, adopts a slightly modified version, where the second many-to-many relationship is replaced by a one-to-one relationship between roles and a new entity named *Access Policy* (Fig. 7 (b)).

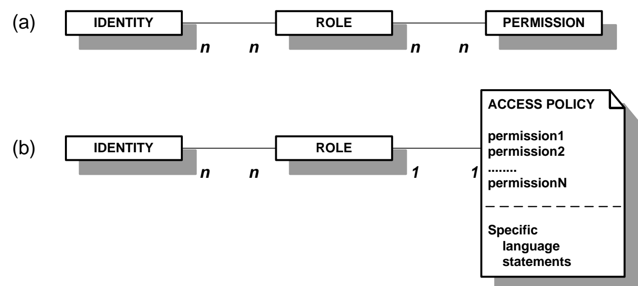

Fig. 7. The standard RBAC model (a) and the UbiMedic implementation (b).

Such an implementation offers the possibility of better expressing permissions in the form of more complex rules using suitable declarative languages, such as Ponder [11] or Drools [4]. This requires the adoption of some related parsers and interpreters to integrate these languages in the system.

The authorization data are stored in a centralized database and the *Authorization Agent* is the only delegated to access and deliver the permissions on request. It provides the administration utilities for the permission update and answers to other agents requests. However, both for safety reasons and for a coherent separation of concerns among the middleware levels, the single agents can't directly access the authorization module, but an intermediate component is always involved. When a Context Agent is created (as

described in the previous Section) the already quoted Session Factory Agent is the responsible for collecting all the required session information for the identity, including the proper permissions (

Fig. 8 (a)). Now the identities' authorization data can be stored in their own Context Agent. For any subsequent authorization request (

Fig. 8 (b)), each identity and application service can interact with other agents through the intercession of the Request Manager Agent and access the permission rules stored in the Context Agents to contact.

9. Critical Evaluation and Future Challenges

Throughout this paper, the architecture of the UbiMedic platform has been outlined, while in this final section we will evaluate the different peculiarities of this framework, compared to the related work.

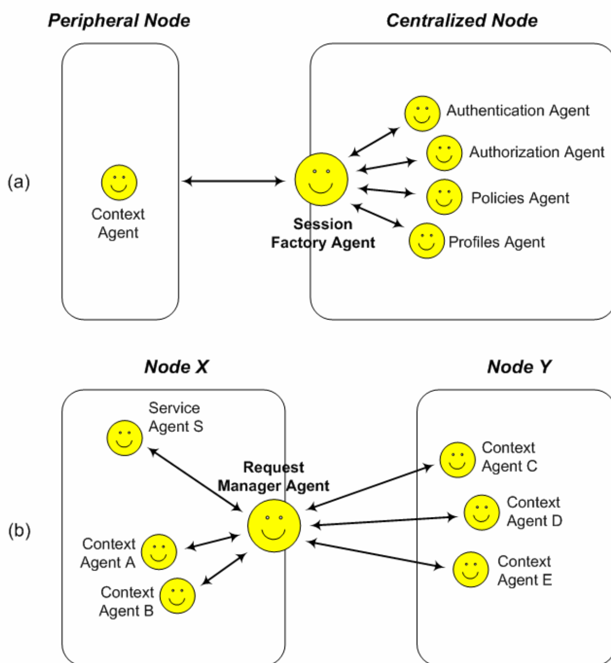

Fig. 8. Agent relations in retrieving authorization data.

Most healthcare systems are focused on some particular application scenarios: e.g., LIFENET provides cardiac emergency services, ILSA deals with monitoring of elderly people, etc. UbiMedic is proposed as a *general* medical emergency platform, to be installed in hospitals, ambulances and other healthcare organizations. In addition, even though UbiMedic has been designed for medical purposes, it is suitable to a wide range of different applications: it can be used in any field requiring a *pervasive*, *dynamic* and *context-aware* system. It has already been stressed the key importance of giving applications an up-to-date view of the situation around them, so that they can take intelligent and informed actions. Situational data are captured by environment-embedded sensors and other mobile devices and are stored in the described Context Agents, available to other entities in the platform.

Thanks to its generality, UbiMedic can *offer the programmer the power of mobile agents, even without requiring an agent-based design*. There are, in fact, many applications that would greatly benefit from an agent-oriented design, in particular those applications where several distributed components have to cooperate towards the achievement of a common goal (e.g., several agents collaborating to find the shortest path to a certain hospital). However, other applications (legacy applications, in particular) do not need the complexity of agents at design-time, although this does not imply that they cannot participate in a MAS: thanks to the Service Agent intermediation, they interact with other services exchanging ACL (Agent Communication Language) messages and are thus fully integrated in the agent platform. The adopted service model has therefore the potential to overcome most of the difficulties that many agent proposals run into (refer to the ILSA project quoted in Section 3).

The other outstanding asset of UbiMedic is its *extensible support for heterogeneous medical devices*, which is by no means the case of the analyzed related work: in these systems the management is hardwired into the platform and can only cope with certain special medical devices (e.g., the Lifepak 12 presented in Section 3). This represents a heavy limitation for the organizations that have to adopt such systems, because they cannot fully exploit their existent instrumentation. The UbiMedic approach to the device integration problem, by means of the three agents described in Section 6.1.1, allows first device manufactures and then programmers to easily integrated their products and make them accessible throughout the distributed platform. Furthermore, even more resource-constrained devices are supported, thanks to the adaptivity and mobility features of the PRIA agent of Section 6.1.1.

Apart from these relevant benefits, there are some other practical issues that have still to be addressed and some problems have to be solved before UbiMedic or a similar middleware could really be used in healthcare commercial applications. Some issues are strictly related to the adopted agent technology [14], while others are more general and inherent to the real world of the application field [9]. An overview of many of these future challenges is given in the following subsections.

9.1 Security

First of all, security is a very important matter: it is a fundamental issue, concerning both information confidentiality and authorized operations. Many of medical information are subject to privacy protection; they often cannot be made public to other people except those directly involved. On the other hand, the requested tasks during a medical and territorial emergency are very delicate and imply high responsibilities. The middleware technology must grant a secure implementation for accessing the information and avoiding unauthorized actions. While inter-agent communications can be encrypted to save confidentiality, the authentication of mobile and autonomous agents could be more difficult. Furthermore, both the agents and the nodes where they are running have to protect themselves, the former against malicious nodes, and the latter against malicious agents [14]. Both the entities, in fact, could potentially try to attack respectively the node to exploit or

damage its resources and the agent to take out its secrets. New technological solutions would be very useful in security issues, but on the other hand a possible way to partially face the problem could be to communicate each sensible piece of information between the agent and a central secure repository node, so that the agent carries no secret information during the migration.

9.2 Communication Standards and Common Ontologies

A system able to integrate different sources and applications has to exploit open and shared protocols. On the other hand, medical devices and other applications should as well be compliant with these protocols so as to be integrated into the system. But both devices and existing medical applications are often closed and proprietary solutions. These topics have been further analyzed in [3], where the concept of *roles* has been proposed as a concrete solution to the device integration problem and to its commercial implications.

Moreover, shared communication standards and complete common medical ontologies, indispensable to achieve effective integration and portability, are not used or do not exist at present.

9.3 Legal Conformity

All healthcare activities are strictly regulated by laws and professional rules, which can differ on the base of local, national and international regulations. Besides, specific rules are defined for electronically conducted activities in substitution of traditional non-technological procedures. In particular, a multi-agent system, under an agent form, can represent official organizations and legal entities cooperating to reach some common goals. All these individuals have to respect some laws and deontological rules, which have to be reflected in the multi-agent system. Finally, many other regulations will be formalized in the future, with the adoption and diffusion of such systems.

9.4 Social and Professional Acceptance

Doctors and other health-care professionals run often into difficulties when new technological solutions are proposed to support their work. It can be hard for people used to work in a traditional or simply different way to adapt themselves to new methodologies, especially if they have no much time to care about these technology aspects. For these reasons, a system supporting medical emergencies must be very practical and user-friendly. But even if this requirement is satisfied, such a pervasive system has to face users' suspiciousness. In fact, both professionals and patients could not just feel safe entrusting their work and confidential information to new technologies.

10. Conclusions and Future Work

In this paper, we have discussed some of the major challenges arising from the application of the software agent paradigm to the healthcare environment, with a focus on medical emergency situations. We found that this paradigm can sometimes reveal itself complex and hard to exploit, but it is capable of offering, nevertheless, powerful features that make it perhaps the ideal fit in this context.

Therefore, we propose the adoption of a middleware, built on top of the JADE platform, in which medical devices, doctors and ambulances interact by means of their representative agents running and interacting in the distributed platform. Even if the presented framework is still a work in progress, we have envisioned the advantages of agent oriented programming in the emergency domain. Unlike traditional paradigms, agents exhibit the property of being autonomous and interactive and, coupled with mobility, they are capable of performing dynamic and intelligent inference tasks during their execution. Adopting a framework based on the software agent paradigm, we can achieve a higher degree of flexibility by allowing applications to dynamically adapt to the changing demands of their execution environments. Application services are, instead, unaware of the agent platform and can be easily deployed as plain Java code, without having to manage all the complex details of agent-hood.

The system development started with the implementation of the lowest services of the UbiMedic framework; they still need to be further detailed in the future, together with a more complete analysis of the other middleware modules. The whole framework will be tested step by step to validate the architectural choices and to identify possible single points of failure.

Acknowledgements

Work supported by the European Community within the EU FET project "CASCADAS".

References

[1] P Bellavista, D Bottazzi, A Corradi and R Montanari, Challenges, opportunities and solutions for ubiquitous eldercare. DEIS University of Bologna Technical Report, October 2005.

[2] G Cabri, F De Mola and R Quitadamo, Supporting a territorial emergency scenario with Services and Agents: a case study comparison. Proceeding of ACEC, WETICE 2006, Manchester, June 2006 (To appear).

[3] G Cabri, F De Mola, R Quitadamo and F Zambonelli, Agent-based integration of medical devices for monitoring purposes. Proceeding of the 4th Workshop of Agent Applied in Health Care, ECAI'06, Riva del Garda, Italy, August 2006 (To appear).

[4] Drools, Open Source project by Bob McWhirter, hosted at The Codehaus http://www.drools.org

[5] A Fuggetta, G P Picco and G Vigna, Understanding code mobility. IEEE Transactions on Software Engineering, Vol. 24, no. 5, May 1998, pp. 342-362.

[6] K Z Haigh, L M Kiff, J Myers, V Guralnik, C W Geib, J Phelps and T Wagner, The independent lifestyle assistant (I.L.S.A.): AI lessons learned. The 16th Innovative Applications of Artificial Intelligence Conference (IAAI'04), San Jose, CA., July 2004, pp. 852-857.

[7] JADE, Java Agents Development Framework, TILAB, Torino, http://jade.tilab.com

[8] Medtronic Physio-Control, http://www.medtronic-ers.com

[9] J Nealon and A Moreno, Agent-based applications in

health care. In: Applications of Software Agent Technology in the Health Care Domain, Whitestein Series in Software Agent Technologies, Birkhäuser Verlag, Basel, Germany, pp. 3-18.
[10] A Padovitz, S W Loke, A Zaslavsky and B Burg, Towards a general approach for reasoning about context, situations and uncertainty in ubiquitous sensing: Putting geometrical intuitions to work. 2nd International Symposium on Ubiquitous Computing Systems (UCS'04), Tokyo, Japan, 2004.
[11] PONDER, Policy Language for Distributed Systems Management Policy, Research Group, Department of Computing, Imperial College, London, http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml
[12] RBAC, Role-based access control. NIST, National Institute of Standards and Technology, http://csrc.nist.gov/rbac
[13] V Shnayder, B Chen, K Lorincz, T R F Fulford-Jones and M Welsh, Sensor networks for medical care. Technical Report TR-08-05, Division of Engineering and Applied Sciences, Harvard University, 2005.
[14] G Vigna, Mobile Agents: Ten reasons for failure. Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04), Berkeley, California, USA, January 2004, pp. 298-299.

Author Bios

Francesco De Mola is a PhD student in Information and Communication Technologies at the University of Modena and Reggio Emilia (Italy). He received the Laurea degree in Computer Engineering from the same University in 2005. His research interests include agent applications in ambient intelligence and telemedicine scenarios.

Giacomo Cabri is an assistant professor in Computer Science at the University of Modena and Reggio Emilia, Italy. He received the Laurea degree in Computer Engineering from the University of Bologna in 1995, and the PhD in Computer Science from the University of Modena and Reggio Emilia in 2000. His affiliation is the Department of Information Engineering at Modena. His research interests include methodologies, tools and environments for agents and mobile computing, wide-scale network applications, and object-oriented programming.

Nicola Muratori is a consultant at O2Studio Ingegneri Associati in Albinea (Reggio Emilia, Italy). He received the Laurea degree in Computer Engineering from the University of Parma in 2000. His works experiences include distributed applications in financial and commercial environments. His research interests include agent applications in telemedicine scenarios.

Raffaele Quitadamo is a PhD Student in Computer Science at the University of Modena and Reggio Emilia. He received the Laurea degree in Computer Engineering from the University of Bologna in 2004. His affiliation is the Department of Information Engineering at Modena (Italy). His research interests include technologies and infrastructures for mobile and pervasive computing, service-oriented computing and autonomic communications.

Franco Zambonelli is an associate professor in Computer Science at the University of Modena and Reggio Emilia, since 2001. He obtained the Laurea degree in Electronic Engineering in 1992, and the PhD in Computer Science in 1997, both from the University of Bologna. In 1996, he obtained a fellowship to do research activity in distributed systems at the Brown University (Rhode Island, USA). In 1999, he has been visiting researcher at the University of Southampton (UK), where he has done research activity in the area of intelligent agents. His current affiliation is the Department of Engineering Methods and Science at Reggio Emilia. His current research interests include: middleware for mobile and embedded systems, agents and mobile agents technologies, agent-oriented software engineering.